8. The CORELIB Module

Created: April 1, 2003
Updated: February 4, 2004

The CORELIB API [Library **xncbi**:include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

The CORELIB provides a portable low-level API and many useful application framework classes for argument processing, diagnostics, environment interface, object and reference classes, portability definitions, portable exceptions, stream wrappers, string manipulation, threads, etc.
This chapter provides reference material for many of CORELIB's facilities. For an overview of CORELIB please refer to the CORELIB section in the introductory chapter on the C++ Toolkit.
NB: The CORELIB must be linked to every executable which uses the NCBI C++ toolkit!

Chapter Outline

The following is an outline of the topics presented in this chapter:

- Writing a Simple Application
    - NCBI C++ Toolkit Application Framework Classes
        - CNcbiApplication
        - CNcbiArguments
        - CNcbiEnvironment
        - CNcbiRegistry
        - CNcbiDiag
    - Creating a Simple Application
        - UNIX Systems

- Using the CNcbiRegistry Class
    - Working with the Registry class: CNcbiRegistry
    - Syntax of the Registry Configuration File
    - Search Order for Initialization (*.ini) Files
    - Setting Persistency and Modifiability of Registry Parameters Using CNcbiRegistry::EFlags
    - Main Methods of CNcbiRegistry
    - Additional Registry Methods
- Portable Stream Wrappers
- Working with Diagnostic Streams (*)
    - Setting Diagnostic Severity Levels
    - Controlling Apperance of Diagnostic Message using Post Flags
    - Defining the Output Stream
    - The Message Buffer
    - Error codes and their Descriptions
        - Preparing an Error Message File
        - Using Error Codes in a Program
    - Defining Custom Handlers using CDiagHandler
    - The ERR_POST Macro
    - The _TRACE macro
    - Example Usage of the CNcbiDiag class
- Debug Macros
- Handling Exceptions

- Defining the Standard NCBI C++ types and their Limits
    - Headers Files containing Portability Definitions
    - Built-in Integral Types
    - Auxiliary Types
    - Fixed-size Integer Types
    - The "Ncbi_BigScalar" Type
    - Encouraged and Discouraged Types
- Understanding Smart Pointers: the CObject and CRef Classes
    - STL auto_ptrs
    - The CRef (*) Class
    - The CObject (*) Class
    - The CObjectFor (*) class: using smart pointers for standard types
    - When to use CRefs and auto_ptrs
    - CRef Pitfalls
        - Inadvertent Object Destruction
- Atomic Counters
- Portable mechanisms for loading DLLs
    - CDll Constructor
    - CDll Basename
    - Other CDll Methods
- Executing Commands and Spawing Processes using the CExec class
    - Executing a System Command using the System() Method
    - Defining Spawned Process Modes (EMode type)

- Spawning a Process using SpawnX() Methods
- Waiting for a Process to Terminate using the Wait() method

- Implementing Parralelism using Threads and Synchronization Mechanisms
  - Using Threads
  - CThread (*) class public methods
  - CThread (*) class protected methods
  - Thread Life Cycle
  - Referencing Thread Objects
  - Thread local storage (CTls<> class [*])
  - Mutexes
    - CMutex
    - CFastMutex
    - SSystemMutex and SSystemFastMutex
    - CMutexGuard and CFastMutexGuard
    - Lock Classes
      - CRWLock
      - CAutoRW
      - CReadLockGuard
      - CWriteLockGuard
      - CInternalRWLock
      - CSemaphore
- Working with File and Directories using CFile and CDir
  - CDirEntry class

    - CFile class
    - CDir class
    - CMemoryFile class
- String APIs
    - String Constants
    - NStr Class
    - UTF Strings
    - PCase and PNocase
- Portable Time Class
    - CTime Class Constructors
    - Other CTime Methods
- Template Utilities
    - Function Objects
    - Template Functions
- Miscellaneous Types and Macros
    - Miscellaneous Enumeration Types
    - AutoPtr Class
    - ITERATE macros
    - Sequence Position Types

Demo Cases [src/app/sample/basic]
Test Cases [src/corelib/test]

Writing a Simple Application

This section discusses how to write a simple application using the CNcbiApplication and related class. A conceptual understanding the uses of the CNcbiApplication and related classes is presented in the introductory chapter on the C++ Toolkit.

This section discusses the following topics:

- Basic classes of NCBI C++ toolkit
- Creating a simple application
- Inside the NCBI Application class

NCBI C++ Toolkit Application Framework Classes

The following five fundamental classes form the foundation of the C++ Toolkit:

- class CNcbiApplication
- class CNcbiArguments (see also CArgDescriptions, CArgs, ...)
- class CNcbiEnvironment
- class CNcbiRegistry
- class CNcbiDiag

Each of these classes are discussed in the following sections:

CNcbiApplication

CNcbiApplication is an abstract class used to define the basic functionality and behavior of an NCBI application. Since this application class effectively supersedes the C-style ***main()*** function, minimally, it must provide the same functionality, i.e.

- A mechanism to execute the actual application
- A data structure for holding program command-line arguments (`"argv"`)
- A data structure for holding environment variables

In addition, the application class provides the same features previously implemented in the C Toolkit, namely:

- Mechanisms for specifying where, when, and how errors should be reported
- Methods for reading, accessing, modifying, and writing information in the application's registry (configuration) file

- Methods to describe, and then automatically parse, validate and access program command-line arguments, and to generate the *USAGE* message

The mechanism to execute the application is provided by ***CNcbiApplication***'s member function ***Run()*** - which you must write your own implementation of. The ***Run()*** function will be automatically invoked by ***CNcbiApplication::AppMain()***, after it has initialized its ***CNcbiArguments, CNcbiEnvironment, CNcbiRegistry***, and ***CNcbiDiag*** data members.

CNcbiArguments

CNcbiArguments class provides a data structure for holding the application's command-line arguments, along with methods for accessing and modifying these. Access to the argument values is implemented using the built-in *[ ]* operator. For example, the first argument in `argv` (following the program name) can be retrieved using the ***CNcbiApplication::GetArguments()*** method:

```
string arg1_value = GetArguments()[1];
```

Here, ***GetArguments()*** returns the ***CNcbiArguments*** object, whose argument values can then be retrieved using the *[ ]* operator. Four additional ***CNcbiArgument*** member functions support retrieval and modification of the program name (initially `argv[0]`). A helper class, described in Processing Command Line Arguments, supports the generation of *USAGE* messages and the imposition of constraints on the values of the input arguments.

In addition to the ***CNcbiArguments*** class, there are other related classes used for argument processing. The ***CArgDescriptions*** and ***CArgDesc*** classes are used for describing unparsed arguments; ***CArgs*** and ***CArgValue*** for parsed argument values; ***CArgException*** and ***CArgHelpException*** for argument exceptions; and ***CArgAllow***, ***CArgAllow_{Strings, ..., Integers, Doubles}*** for argument constraints. These classes are discussed in the section on Processing Command Line Arguments.

When using C++ Toolkit on Mac OS you can specify command-line arguments in a separate file with the name of your executable and ".args" extension. Each argument should be on a separate line (see Table 1).

Table 1. Example of Command-line Arguments

Command-line parameters	File Content
-gi "Integer" (GI id of the Seq-Entry to examine) OPTIONAL ARGUMENTS: -h (Print this USAGE message; ignore other arguments) -reconstruct (Reconstruct title) -accession (Prepend accession) -organism (Append organism name)	-gi 10200 -reconstruct -accession -organism

Please note: File must contain Macintosh-style line breaksNo extra spaces are allowed after argument ("-accession" and not "-accession ")Arguments must be followed by an empty terminating line.

CNcbiEnvironment

The ***CNcbiEnvironment*** class provides a data structure for storing, accessing, and modifying the environment variables accessed by the C library routine ***getenv()***.

The following describes the public interface to the ***CNcbiEnvironment***:

```
class  CNcbiEnvironment
{
public:
    /// Constructor.
    CNcbiEnvironment(void);

    /// Constructor with the envp parameter.
    CNcbiEnvironment(const char* const* envp);

    /// Destructor.
    virtual ~CNcbiEnvironment(void);

    /// Reset environment.
    ///
    /// Delete all cached entries, load new ones from "envp" (if not NULL).
    void Reset(const char* const* envp = 0);

    /// Get environment value by name.
    ///
    /// If environmnent value is not cached then call "Load(name)" to load
    /// the environmnent value.  The loaded name/value pair will then be
    /// cached, too, after the call to "Get()".
    const string& Get(const string& name) const;

};
```

For example, to retrieve the value of environment variable `PATH`:

```
string arg1_value = GetEnvironment().Get("PATH");
```

In this example, the ***GetEnvironment()*** is defined in the ***CNcbiApplication*** class and returns the ***CNcbiEnvironment*** object for which the ***Get()*** method is called with the environment variable `PATH`

To delete all of the cached entrues and reload new ones from the environment pointer(envp), use the ***CNcbiEnvironment::Reset()*** method.

CNcbiRegistry

The ***CNcbiRegistry*** class is used to load, access, modify and store runtime information read from a configuration file. Previously, these files were by convention named *.*rc* files on `UNIX` systems. The convention for all platforms now is to name such files *.*ini* (where * is by default the application name).

The name of configuration file is specified by argument `conf` of CNcbiApplication::AppMain() (see Table 2).

Table 2. Location of Configuration Files

conf	Where to look for the config file
empty [default]	Compose config file name from the base application name plus *.ini*. Also try to strip file extensions, e.g. for the application named `my_app.cgi.exe` try subsequently: *my_app.cgi.exe.ini*, *my_app.cgi.ini*, *my_app.ini*. Using these names, search in directories as described in the "Otherwise" case for non-empty `conf` (see below).
NULL	Do not even try to load registry at all
non-empty	If `conf` contains a path, then try to load from the config file named `conf` (only and exactly!) If the path is not fully qualified, and it starts from *../* or *./*, then look for the config file starting from the current working dir **Otherwise** (only a basename, without path), the config file will be searched for in the following places (in the order of preference): 1. current work directory 2. directory defined by environment variable `NCBI` 3. user home directory 4. program directory

On success, you can access the loaded configuration (registry) using method CNcbiApplication::GetConfig(). Application will throw an exception if the config.file is found, it is not empty, and either cannot be opened or contains invalid entries. If `conf` is not *NULL*, and the config file cannot not be found, then a warning will be posted to the application diagnostic.

Additional details on the ***CNcbiRegistry*** can be found in the section on The CNcbiRegistry Class.

CNcbiDiag

The ***CNcbiDiag*** class implements much of the functionality of the NCBI C Toolkit error processing mechanisms. Each instance of ***CNcbiDiag*** has a private buffer to handle a single message, along with private severity level and post flags and their associated get/set methods. A CNcbiDiag object has the look and feel of an output stream; its member functions and friends include output operators >> and format manipulators. The default is to post errors to `stderr`, with the action determined by the severity level of the message, however user can provide another stream to post to, or create an arbitrary callback to do the job, or just ignore all diagnostics. See also the sections on Diagnostic Streams and Message Posting.

Creating a Simple Application

This section discusses the following topics:

- UNIX Systems

- MS Windows
- Discussion of the Sample Application

UNIX Systems

Using ***new_project.sh*** shell script, create a new project *sample* in the folder *sample*:

```
$NCBI/c++/scripts/new_project.sh sample app
```

This will create:

1. the project folder - *sample*,
2. the source file - *sample*.cpp,
3. the makefile - Makefile.*sample*_app.

Then build the project and run the application:

```
cd sample; make -f Makefile.sample_app; ./sample
```

MS Windows

1. In Microsoft Visual Studio create a new project/workspace: choose *Win32 Console Application*; then supply a Project name (such as Sample) and select OK; then choose *An empty project* and select Finish.
2. Copy the sample source file into the project directory, rename it, then add to the project.
3. Modify the project settings:
   - Enable Run-time type information (in *C/C++* - *C++ language*),
   - Disable using precompiled headers (in *C/C++* - *Precompiled headers*),
   - Add additional include directory (in *C/C++* - *Preprocessor*). Here, at NCBI, it could be *Dizzy**public**cxx**include*, that is, the "root" of all includes,
   - Add additional library path (*Link-Input*). Here, at NCBI it could be *Dizzy**public**cxx**Debug*. Please note, this library path is configuration-dependent, that is it must be different for each configuration you are going to build the project in,
   - Remove all standard libraries in *Link-Input-Object/library modules*,
   - Add `xncbi.lib` NCBI library to the project.
4. Build the project and run the application.

Discussion of the Sample Application

In the sample application above:

1. There is an application class derived from ***CNcbiApplication***, which overrides purely virtual function ***Run()***, and also initialization (***Init()***) and cleanup (***Exit()***) functions:

```
class CSampleApplication : public CNcbiApplication
{
private:
    virtual void Init(void);
    virtual int Run(void);
    virtual void Exit(void);
};
```

2. Program's main function creates an object of the application class and calls its ***AppMain ()*** function:

```
int main(int argc, const char* argv[])
{
    CSampleApplication theApp;
    // Execute main application function
    theApp.AppMain(argc, argv, 0, eDS_Default, 0);
}
```

3. Application's initialization function creates argument descriptions object, which describes the expected command line arguments and the usage context:

```
void CSampleApplication::Init(void)
{
    // Create command-line argument descriptions
    auto_ptr<CArgDescriptions> arg_desc(new CArgDescriptions);
    // Specify USAGE context
    arg_desc->SetUsageContext(GetArguments().GetProgramBasename(), "CArgDescrip-
tions demo program");
    ...
    // Setup arg.descriptions for this application SetupArgDescriptions(arg_desc.
release());
}
```

4. Application's ***Run()*** function prints those arguments into the standard output stream, or in a file.

More realistic examples of applications, which utilize NCBI C++ toolkit can be found here.

Inside the NCBI Application class

Here is a somewhat simplified view of the application's class definition:

```
class CNcbiApplication
{
public:
```

```
/// Main function (entry point) for the NCBI application.
///
/// You can specify where to write the diagnostics to (EAppDiagStream),
/// and where to get the configuration file (LoadConfig()) to load
/// to the application registry (accessible via GetConfig()).
///
/// Throw exception if:
///  - not-only instance
///  - cannot load explicitly specified config.file
///  - SetupDiag() throws an exception
///
/// If application name is not specified a default of "ncbi" is used.
/// Certain flags such as -logfile, -conffile and -version are special so
/// AppMain() processes them separately.
/// @return
///   Exit code from Run(). Can also return non-zero value if application
///   threw an exception.
/// @sa
///   Init(), Run(), Exit()
int AppMain(int argc, const char **argv, const char **envp,
            EAppDiagStream diag, const char* config, const string& name);

/// Initialize the application.
///
/// The default behavior of this is "do nothing". If you have special
/// initialization logic that needs to be peformed, then you must override
/// this method with your own logic.
virtual void Init(void);

/// Run the application.
///
/// It is defined as a pure virtual method -- so you must(!) supply the
/// Run() method to implement the application-specific logic.
/// @return
///   Exit code.
virtual int  Run(void) = 0;

/// Cleanup on application exit.
///
/// Perform cleanup before exiting. The default behavior of this is
/// "do nothing". If you have special cleanup logic that needs to be
/// performed, then you must override this method with your own logic.
virtual void Exit(void);

/// Get the application's cached unprocessed command-line arguments.
const CNcbiArguments& GetArguments(void) const;

/// Get parsed command line arguments.
///
/// Get command line arguments parsed according to the arg descriptions
/// set by SetArgDescriptions(). Throw exception if no descriptions
/// have been set.
/// @return
```

```
    ///   The CArgs object containing parsed cmd.-line arguments.
    /// @sa
    ///   SetArgDescriptions().
    const CArgs& GetArgs(void) const;

    /// Get the application's cached environment.
    const CNcbiEnvironment& GetEnvironment(void) const;

    /// Get the application's cached configuration parameters.
    const CNcbiRegistry& GetConfig(void) const;

    /// Flush the in-memory diagnostic stream (for "eDS_ToMemory" case only).
    ///
    /// In case of "eDS_ToMemory", the diagnostics is stored in
    /// the internal application memory buffer ("m_DiagStream").
    /// Call this function to dump all the diagnostics to stream "os" and
    /// purge the buffer.
    /// @param  os
    ///   Output stream to dump diagnostics to. If it is NULL, then
    ///   nothing will be written to it (but the buffer will still be purged).
    /// @param  close_diag
    ///   If "close_diag" is TRUE, then also destroy "m_DiagStream".
    /// @return
    ///   Total number of bytes actually written to "os".
    SIZE_TYPE FlushDiag(CNcbiOstream* os, bool close_diag = false);

    /// Get the application's "display" name.
    ///
    /// Get name of this application, suitable for displaying
    /// or for using as the base name for other files.
    /// Will be the 'name' argument of AppMain if given.
    /// Otherwise will be taken from the actual name of the application file
    /// or argv[0].
    string GetProgramDisplayName(void) const;

protected:

    /// Setup application specific diagnostic stream.
    ///
    /// Called from SetupDiag when it is passed the eDS_AppSpecific parameter.
    /// Currently, this calls SetupDiag(eDS_ToStderr) to setup diagonistic
    /// stream to the std error channel.
    /// @return
    ///   TRUE if successful, FALSE otherwise.
   virtual bool SetupDiag_AppSpecific(void);

    /// Load configuration settings from the configuration file to
    /// the registry.
    ///
    /// Load (add) registry settings from the configuration file specified as
    /// the "conf" arg passed to AppMain(). The "conf" argument has the
    /// following special meanings:
    ///  - NULL      -- dont even try to load registry from any file at all;
```

```
    ///  - non-empty -- if "conf" contains a path, then try to load from the
    ///                 conf.file of name "conf" (only!). Else - see NOTE.
    ///                 TIP: if the path is not fully qualified then:
    ///                      if it starts from "../" or "./" -- look starting
    ///                      from the current working dir.
    ///  - empty     -- compose conf.file name from the application name
    ///                 plus ".ini". If it does not match an existing
    ///                 file, then try to strip file extensions, e.g. for
    ///                 "my_app.cgi.exe" -- try subsequently:
    ///                   "my_app.cgi.exe.ini", "my_app.cgi.ini", "my_app.ini".
    ///
    /// NOTE:
    /// If "conf" arg is empty or non-empty, but without path, then config file
    /// will be sought for in the following order:
    ///  - in the current work directory;
    ///  - in the dir defined by environment variable "NCBI";
    ///  - in the user home directory;
    ///  - in the program dir.
    ///
    /// Throw an exception if "conf" is non-empty, and cannot open file.
    /// Throw an exception if file exists, but contains invalid entries.
    /// @param reg
    ///   The loaded registry is returned via the reg parameter.
    /// @param conf
    ///   The configuration file to loaded the registry entries from.
    /// @return
    ///   TRUE only if the file was non-NULL, found and successfully read.
   virtual bool LoadConfig(CNcbiRegistry& reg, const string* conf);
   ..............
};
```

The ***AppMain()*** function is also inherited from the parent class. Although this function accepts up to six arguments, this example passes only the first two, with missing values supplied by defaults. The remaining four arguments specify:

- (#3) a NULL-terminated array of '\0'-terminated character strings from which the environment variables can be read
- (#4) how to setup a diagnostic stream for message posting
- (#5) the name of a *.ini* configuration file (see above for its default location)
- (#6) a program name (to be used in lieu of `argv[0]`)

In order to avoid the display of a warning message when no configuration file is present, the *.ini* file should be explicitly specified as *NULL* (zero), as in:

```
AppMain (argc, argv, envp, diag_stream, 0, progname);
```

AppMain() begins by resetting the internal data members with the actual values provided by the arguments of ***main()***. Once these internal data structures have been loaded, ***AppMain()*** calls the virtual functions ***Init(), Run(),*** and ***Exit()*** in succession to execute the application.

The ***Init()*** and ***Exit()***virtual functions are provided as places for developers to add their own methods for specific applications. As this example does not require any additional initialization/ termination, these two functions are empty. The ***Run()*** method prints out the message defined in *justApp.cpp* and exits.

The ***FlushDiag()*** method is useful if the diagnostic stream has been set to `eDS_toMemory` which means that diagnostic messages are stored in an internal application memory buffer. You can then call ***FlushDiag()*** to output the stored messages on the specified output stream. The method will also return the number of bytes written to the output stream. If you specify NULL for the output stream, the memory buffers containing the diagnostic messages will be purged but not deallocated, and nothing will be written to the output. If the close_diag parameter to ***FlushDiag()*** is set to TRUE, then the memory buffers will be deallocated (and purged, of course).

The ***GetProgramDisplayName()*** method simply returns the name of the running application, suitable for displaying in reports or for using as the base name for building other related file names.

The protected virtual function ***SetupDiag_AppSpecific()*** can be redefined to setup error posting specific for your application. ***SetupDiag_AppSpecific()*** will be called inside ***AppMain()*** by default if the error posting has not been setup already. Also, if you pass *diag = eDS_AppSpecific* to ***AppMain()***, then ***SetupDiag_AppSpecific()*** will be called for sure, regardless of the error posting setup that was active before the ***AppMain()*** call.

The protected virtual function ***LoadConfig()***reads the program's *.ini* configuration file to load the application's parameters into the registry. The default implementation of ***LoadConfig()*** expects to find a configuration file named *progname.ini*, and will generate a warning to the user if no such file is found.

The NCBI application (application built by deriving from ***CNcbiApplication***) throws the exception the ***CAppException*** when any of the following conditions are true:

- Command line argument description cannot be found and argument descriptions have not been disabled (via call to protected method ***DisableArgDescription().***
- Application diagnostic stream setup has failed.
- Registry data failed to load from a specfied configuration file.
- An attempt is made to create a second instance of the CNcbiApplication -- at any time only one instance can be running.
- The specified configuration file cannot be opened.

As shown above, source files which utilize the ***CNcbiApplication*** class must #*include* the header file where that class is defined, *corelib/ncbiapp.hpp*, in the *include/* directory. This header file in turn includes *corelib/ncbistd.hpp*, which should **always** be #*include*'d.

Processing Command Line Arguments

This section discusses the classes that are used to process command line arguments. A conceptual overview of these classes is covered in an introductory section. This section discusses these classes in details and gives sample programs that use these classes.

This section discusses the following topics:

- Capabilities of the Command Line API
- The Relationships Between the CArgDescriptions, CArgs, and CArgValue classes
- Command Line Syntax
- The CArgDescriptions class
- The CArgs class: A container class for CArgValue objects
- CArgValue class: The Internal Representation of Argument Values
- Code Examples

Capabilities of the Command Line API

The set of classes for argument processing implement automated command line parsing. Specifically, these classes allow the developer to:

- specify attributes of expected arguments, such as name, synopsis, comment, data type, etc.,
- validate values of the arguments passed to the program against these specifications,
- validate the number of positional arguments in the command line,
- generate a USAGE message based on the argument descriptions (**NOTE:**-*h* flag to print the *USAGE* is defined by default),
- access the input argument values specifically type-cast according to their description.

Normally, ***CArgDescriptions*** object that contains the argument description is required and should be created in application's ***Init()*** function before any other initialization. Otherwise ***CNcbiApplication*** creates a default one, which allows any program which utilizes NCBI C++ toolkit to provide some *standard* command line options, namely:

- to obtain a general description of the program as well as description of all available command line parameters (*-h* flag),
- to redirect the program's diagnostic messages into a specified file (*-logfile* key),
- to read the program's configuration data from a specified file (*-conffile* key).

See Table 3 for the standard command line options for the default instance of CArgDescriptions.

Table 3. Standard command line options for the default instance of CArgDescriptions

Flag	Description	Example
-h	Print description of the application's command line parameters.	theapp -h
-logfile	Redirect program's log into the specified file	theapp -logfile theapp_log
-conffile	Read the program's configuration data from the specified file	theapp -conffile theapp_cfg

To avoid creation of a default ***CArgDescriptions*** object which may not be needed for instance if the standard flags described in Table 3 are not used, one should call ***CNcbiApplication::DisableArgDescriptions()*** function from an application object constructor.

Also, it is possible to use ***CNcbiApplication::HideStdArgs(THideStdArgs hide_mask)*** method to hide description of the standard arguments (*-h, -logfile, -conffile*) in the USAGE message. Please note, this only hides the description of these flags - it is still possible to use them.

The Relationships Between the *CArgDescriptions*, *CArgs*, and *CArgValue* classes

The ***CArgDescriptions*** class provides an interface to describe the data type and attributes of command-line arguments, via a set of ***AddXxx()*** methods. Additional constraints on the argument values can be imposed using the ***SetConstraint()*** method. The ***CreateArgs()*** method is passed the values of all command line arguments at runtime. This method verifies their overall syntactic structure and matches their values against the stored descriptions. If the arguments are parsed successfully, a new ***CArgs*** object is returned by ***CreateArgs()***.

The resulting ***CArgs*** object will contain parsed, verified and ready-to-use argument values which are stored as ***CArgValue***'s. The value of a particular argument can be accessed using the argument's name (as specified in the ***CArgDescriptions*** object), and the returned ***CArgValue*** object can then be safely type-cast to a correct C++ type (***int***, ***string***, ***stream***, etc.) because the argument types have been verified. These class relations and methods can be summarized schematically as shown in Figure 1.

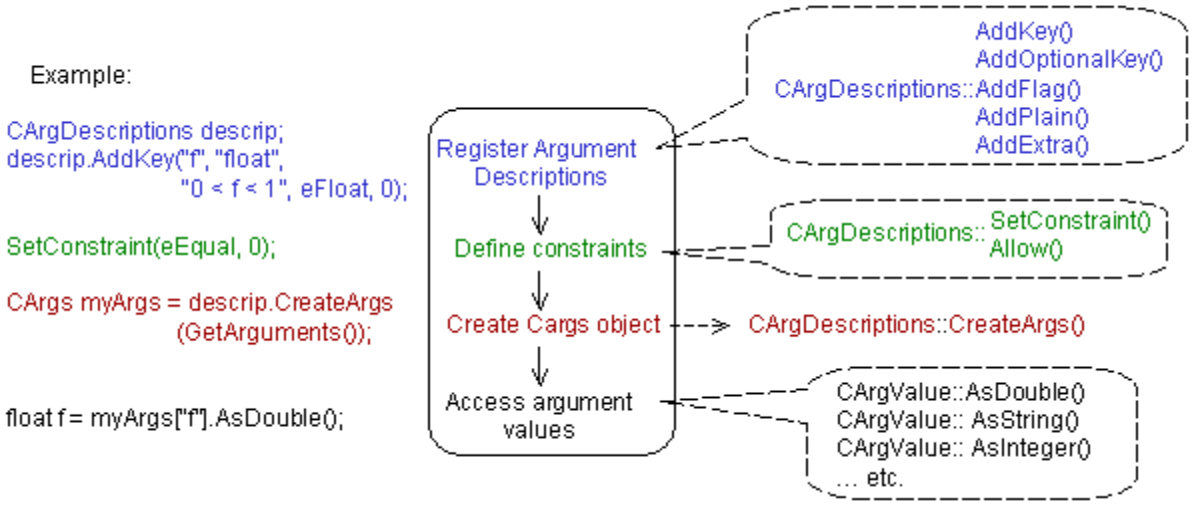

Figure 1: Argument Processing Class Relations

The last statement in this example implicitly references a ***CArgValue*** object, in the value returned when the *[ ]* operator is applied to `myArgs`. The method ***CArgValue::AsDouble()*** is then applied to this object to retrieve a ***double***.

Command Line Syntax

This API can describe and work with command line arguments that fit the following profile:

```
// progname
// {arg_key, arg_key_opt, arg_key_dflt, arg_flag} [--]
// {arg_pos} {arg_pos_opt, arg_pos_dflt}
// {arg_extra} {arg_extra_opt}
//
// where:
// arg_key := -<key> <value> -- (mandatory)
// arg_key_opt := [-<key> <value>] -- (optional, without default value)
// arg_key_dflt := [-<key> <value>] -- (optional, with default value)
// arg_flag := -<flag> -- (always optional)
// -- := optional delimiter to indicate the beginning of pos. args
// arg_pos := <value> -- (mandatory)
// arg_pos_opt := [<value>] -- (optional, without default value)
// arg_pos_dflt := [<value>] -- (optional, with default value)
// arg_extra := <value> -- (dep. on the constraint policy)
// arg_extra_opt := [<value>] -- (dep. on the constraint policy)
//
// and:
// <key> must be followed by <value>
// <flag> and <key> are case-sensitive, and they can contain
// only alphanumeric characters
// <value> is an arbitrary string (additional constraints can
// be applied in the argument description, see "EType")
//
```

```
// {arg_pos***} and {arg_extra***} -- position-dependent arguments, with
// no tag preceding them.
// {arg_pos***} -- have individual names and descriptions (see methods
// AddPositional***).
// {arg_extra***} have one description for all (see method AddExtra).
// User can apply constraints on the number of mandatory and optional
// {arg_extra***} arguments.
```

Examples:

```
MyProgram1 -reverse -depth 5 -name Lisa -log foo.log 1.c 2.c 3.c
MyProgram2 -i foo.txt -o foo.html -color red
MyProgram3 -a -quiet -pattern 'Error:' bar.txt
```

The *CArgDescriptions* (%20) class

The ***CArgDescriptions*** contains a description of unparsed arguments -- that is, user specified descriptions that are then used to parse the arguments. ***CArgDescriptions*** is used as a container to store the command-line argument descriptions. The argument descriptions are used for parsing and verifying actual command-line arguments.

The following is a list of topics discussed in this section:

- The CArgDescriptions Constructor
- Describing Argument Attributes
- Argument Types
- Restricting the Input Argument Values
- Implementing User-Defined Restrictions Using the CArgAllow class
- Using CArgDescriptions in Applications
- Generating a USAGE Message

The CArgDescriptions Constructor

The constructor for CArgDescriptions excepts a boolean argument, auto_help, set to TRUE by default.

```
CArgDescriptions(bool auto_help = true);
```

If "auto_help" is passed TRUE, then a special flag "-h" will be added to the list of accepted arguments, and passing "-h" in the command line will printout USAGE and ignore all other passed arguments.

Describing Argument Attributes

CNcbiArguments contain many methods named, ***AddXxx()***. The "Xxx" refer to the types of arguments such as mandatory key (named) arguments, optional key arguments, positional arguments, flag arguments, etc. For example, the ***AddKey()*** method refers to adding a description for a mandatory key argument. .

The methods ***AddXxx()*** are passed the following argument attributes: *name* The string that will be used to identify the variable, as in: `CArgs[name].` For all tagged variables in a command line, name is also the key (or flag) to be used there, as in: "*-name value*" (or "*-name*"). *synopsis* (for ***key_****** arguments only)The automatically generated *USAGE* message includes argument description in the format: *-name [synopsis] <type, constraint>* comment *comment* To be displayed in the *USAGE* message, as described above. *value type* One of the scalar values defined in the enumeration, which defines the type of the argument. *default* (for ***key_dflt*** and ***pos_dflt*** arguments only) A default value to be used if the argument is not included in the command line - only available for optional program arguments. *flags* The *flags* argument, which occurs optionally in all but the ***AddFlag()*** method, has meaning only when EType is `eInputFile` or `eOutputFile`, and restricts the mode in which the file should be opened.

Argument Types

The ***CArgDescriptions*** class allows to register command-line arguments that fit one of the following pattern types:

Mandatory named arguments:-<*key*> <*value*> (example: *-age 31*) Position-independent arguments that **must** be present in the command line. ***AddKey(key, synopsis, comment, value_type, flags)***

Optional named arguments:*[-<key> <value>]* (example: *-name Lisa*) Position-independent arguments that are **optional**. ***AddOptionalKey(key, synopsis, comment, value_type, flags)*** A default value can be specified in the argument's description to cover those cases where the argument does not occur in the command line. ***AddDefaultKey(key, synopsis, comment, value_type, default_value, flags)***

Optional named flags:*[-<flag>]* (example:) Position-independent boolean (without value) arguments. These arguments are **always** optional. ***AddFlag(flag, comment, set_value)***

Mandatory named positional arguments:<*value*> (example: *12 Feb*) These are position-*dependent* arguments (of any type), which are read using a *value* only. They do however, have names stored with their descriptions, which they are associated with in an order-dependent fashion. Specifically, the order in which untagged argument descriptions are added to the ***CArgDescriptions*** object using ***AddPositional()*** defines the order in which these arguments should appear in the command line. ***AddPositional(key, comment, value_type, flags)***

Optional named positional arguments:*[value]* (example: *foo.txt bar*) Position-*dependent* arguments that are *optional*. They always go after the *mandatory* positional arguments. The order in which untagged argument descriptions are added to the ***CArgDescriptions*** object using ***Add[Optional|Default]Positional()*** defines the order in which these arguments should appear in the command line. ***AddOptionalPositional(key, comment, value_type, flags)AddDefaultPositional(key, comment, value_type, default_value, flags)***

Unnamed positional arguments (all of the same type: <*value1*> | *[valueN]* (example: *foo.c bar.c xxx.c*) These are also position-*dependent* arguments that are read using a *value* only. They are expected to appear at the very end of the command line, after all named arguments. Unlike the previous argument type however, these arguments do not have individual named descrip-

tions, but share a single "unnamed" description. You can specify how many mandatory and how many optional arguments to expect using `n_mandatory` and `n_optional` parameters: AddExtra(n_mandatory, n_optional, comment, type, flags)

Any of the registered descriptions can be tested for existence and/or deleted using the following ***CArgDescriptions***methods:

```
bool Exist(const string& name) const;
void Delete(const string& name);
```

These methods can also be applied to the unnamed positional arguments (as a *group*), using: *Exist(kEmptyStr)* and *Delete(kEmptyStr).*

Restricting the Input Argument Values

Although each argument's input value is initially loaded as a simple character string, the argument's specified type implies a restricted set of possible values. For example, if the type is `eInteger`, then any integer value is acceptable, but floating point and non-numerical values are not. The EType enumeration quantifies the allowed types and is defined as:

```
    /// Available argument types.
    enum EType {
        eString = 0, ///< An arbitrary string
        eBoolean,    ///< {'true', 't', 'false', 'f'},  case-insensitive
        eInteger,    ///< Convertible into an integer number (int)
        eDouble,     ///< Convertible into a floating point number (dou-
ble)
        eInputFile,  ///< Name of file (must exist and be readable)
        eOutputFile, ///< Name of file (must be writeable)

        k_EType_Size ///< For internal use only
    };
```

Implementing User-Defined Restrictions Using the *CArgAllow* class

It may be neccessary to specify a restricted range for argument values. For example, an integer argument that has a range between 5 and 10. Further restrictions on the allowed values can be specified using the ***CArgDescriptions::SetConstraint()*** method with the ***CArgAllow*** class. For example:

```
auto_ptr<CArgDescriptions> args(new CArgDescriptions);
// add descriptions for "firstint" and "nextint" using AddXxx( ...)
...
CArgAllow* constraint = new CArgAllow_Integers(5,10);
args->SetConstraint("firstInt", constraint);
args->SetConstraint("nextInt", constraint);
```

This specifies that the arguments named *"firstInt"* and *"nextInt"* must both be in the range [5, 10].

The ***CArgAllow_Integers*** class is derived from the **abstract*CArgAllow*** class. The constructor takes the two integer arguments as lower and upper bounds for allowed values. Similarly, the ***CArgAllow_Doubles*** class can be used to specify a range of allowed floating point values. For both classes, the order of the numeric arguments does not matter, as the constructors will use min/max comparisons to generate a valid range.

A third class derived from the ***CArgAllow*** class is the ***CArgAllow_Strings*** class. In this case, the set of allowed values can not be specified by a ***range***, but the following construct can be used to enumerate all eligible string values:

```
CArgAllow* constraint = (new CArgAllow_Strings())->
                            Allow("this)->Allow("that")->Allow("etc");
args.SetConstraint("someString", constraint);
```

Here, the constructor takes no arguments, and the ***Allow()*** method returns `this`. Thus, a list of allowed strings can be specified by daisy-chaining a set of calls to ***Allow()***. A bit unusual yet terser notation can also be used by engaging the comma operator, as in:

```
args.SetConstraint("someString",
                   &(*new CArgAllow_Strings, "this", "that", "etc"));
```

There are two other pre-defined constraint classes:`CArgAllow_Symbols` and `CArgAllow_String`. If the value provided on the command line is not in the allowed set of values specified for that argument, then an exception will be thrown. This exception can be caught and handled in the usual manner, as described in the discussion of Generating a *USAGE* message.

Using *CArgDescriptions* in Applications

The description of program arguments should be provided in application's ***Init()*** function before any other initialization. A good idea is also to specify the program's description here:

```
auto_ptr<CArgDescriptions> arg_desc(new CArgDescriptions);
arg_desc->SetUsageContext(GetArguments().GetProgramBasename(),
                          "program's description here");
// Define arguments, if any
...
SetupArgDescriptions(arg_desc.release());
```

The ***SetUsageContext()*** method is used to define the name of the program and its description, which is to be displayed in the *USAGE* message. As long as an applications's initialization is completed and there is still no argument description, ***CNcbiApplication*** class provides a "default" one. This behavior can be overridden by calling ***DisableArgDescriptions()*** method of *CNcbiAppliation*.

Generating a *USAGE* Message

One of the functions of ***CArgDescriptions*** object is to generate a *USAGE* message automatically (this gives yet another reason to define one). Once such object is defined, there is nothing else to worry about - ***CNcbiApplication*** will do the job for you: ***SetupArgDescriptions()*** method includes parsing the command line and matching arguments against their descriptions. Should an error occurs, - e.g. a mandatory argument is missing, - the program prints a message explaining what was wrong and terminates. The output in this case might look like this:

```
USAGE
  myApp -h -k MandatoryKey [optarg]

DESCRIPTION
   myApp test program

REQUIRED ARGUMENTS
-k <String>
   This is a mandatory alpha-num key argument

OPTIONAL ARGUMENTS
-h
   Print this USAGE message;  ignore other arguments
optarg <File_Out>
   This is an optional named positional argument without default value
================================================================================
"E:\cxx\src\corelib\ncbiapp.cpp", line 437: Error:
NCBI C++ Exception:
    "E:\cxx\src\corelib\ncbiargs.cpp", line 621: Error: (CArgException::eNoArg) Argument
"k". Required argument missing:  `String'
    "E:\cxx\src\corelib\ncbiapp.cpp", line 383: Error: (CArgException::eNoArg) Applica-
tion's initialization failed
```

The message shows a description of the program, and a summary of each argument. In this example, the input file argument's description was defined as:

```
arg_desc->AddKey( "k", "MandatoryKey",
                  "This is a mandatory alpha-num key argument",
                  CArgDescriptions::eString);
```

The information generated for each argument is displayed in the format:

> *me [synopsis] <type [, constraint] > comment [default = ...]*

The CArgs (%20) class: A container class for CArgValue (*) objects

The ***CArgs*** class provides a data structure where the parsed arguments' values can be stored, and includes access routines in its public interface. Argument values are obtained from the unprocessed command-line arguments via the ***CNcbiArguments*** class and then verified and processed according to the argument descriptions defined by user in ***CArgDescriptions***. The following describes the public interface methods in ***CArgs***:

```
class  CArgs
{
public:
    /// Constructor.
    CArgs(void);

    /// Destructor.
    ~CArgs(void);

    /// Check existence of argument description.
    ///
    /// Return TRUE if arg 'name' was described in the parent CArgDescriptions.
    bool Exist(const string& name) const;

    /// Get value of argument by name.
    ///
    /// Throw an exception if such argument does not exist.
    /// @sa
    ///   Exist() above.
    const CArgValue& operator[] (const string& name) const;

    /// Get the number of unnamed positional (a.k.a. extra) args.
    size_t GetNExtra(void) const { return m_nExtra; }

    /// Return N-th extra arg value,  N = 1 to GetNExtra().
    const CArgValue& operator[] (size_t idx) const;

    /// Print (append) all arguments to the string 'str' and return 'str'.
    string& Print(string& str) const;

    /// Add new argument name and value.
    ///
    /// Throw an exception if the 'name' is not an empty string, and if
    /// there is an argument with this name already.
    ///
    /// HINT: Use empty 'name' to add extra (unnamed) args, and they will be
    /// automagically assigned with the virtual names: '#1', '#2', '#3', etc.
    void Add(CArgValue* arg);

    /// Check if there are no arguments in this container.
    bool IsEmpty(void) const;

};
```

The CArgs object is created by executing the ***CArgDescriptions::CreateArgs()*** method. What happens when the ***CArgDescriptions::CreateArgs()*** method is executed is that the command line's arguments are validated against the registered descriptions, and a ***CArgs*** object is created. Each argument value is internally represented as a ***CArgValue*** object, and is added to a container managed by the ***CArgs*** object.

All *named* arguments can be accessed using the *[ ]* operator, as in: *myCArgs["f"]*, where *"f"* is the name registered for that argument. There are two ways to access the **N**-th *unnamed* positional argument: *myCArgs["#N"]* and *myCArgs[N]*, where 1 <= **N** <= GetNExtra().

CArgValue (%20) class: The Internal Representation of Argument Values

The internal representation of an argument value - as it is stored and retrieved from its ***CArgs*** container - is an instance of a ***CArgValue***. The primary purpose of this class is to provide type-validated loading through a set of ***AsXxx()*** methods where "***Xxx***" is the argument type such as "Integer", "Boolean", "Double" etc. The following describes the public interface methods in ***CArgValue***:

```
class  CArgValue : public CObject
{
public:
    /// Get argument name.
    const string& GetName(void) const { return m_Name; }

    /// Check if argument holds a value.
    ///
    /// Argument does not hold value if it was described as optional argument
    /// without default value, and if it was not passed a value in the command
    /// line.  On attempt to retrieve the value from such "no-value" argument,
    /// exception will be thrown.
    virtual bool HasValue(void) const = 0;
    operator bool (void) const { return  HasValue(); }
    bool operator!(void) const { return !HasValue(); }

    /// Get the argument's string value.
    ///
    /// If it is a value of a flag argument, then return either "true"
    /// or "false".
    /// @sa
    ///   AsInteger(), AsDouble(), AsBoolean()
    virtual const string& AsString(void) const = 0;

    /// Get the argument's integer value.
    ///
    /// If you request a wrong value type, such as a call to "AsInteger()"
    /// for a "boolean" argument, an exception is thrown.
    /// @sa
    ///   AsString(), AsDouble, AsBoolean()
    virtual int    AsInteger(void) const = 0;
```

```
    /// Get the argument's double value.
    ///
    /// If you request a wrong value type, such as a call to "AsDouble()"
    /// for a "boolean" argument, an exception is thrown.
    /// @sa
    ///   AsString(), AsInteger, AsBoolean()
    virtual double AsDouble (void) const = 0;

    /// Get the argument's boolean value.
    ///
    /// If you request a wrong value type, such as a call to "AsBoolean()"
    /// for a "integer" argument, an exception is thrown.
    /// @sa
    ///   AsString(), AsInteger, AsDouble()
    virtual bool   AsBoolean(void) const = 0;

    /// Get the argument as an input file stream.
    virtual CNcbiIstream& AsInputFile (void) const = 0;

    /// Get the argument as an output file stream.
    virtual CNcbiOstream& AsOutputFile(void) const = 0;

    /// Close the file.
    virtual void CloseFile (void) const = 0;


};
```

Each of these ***AsXxx()*** methods will access the string storing the requested argument's value, and attempt to convert that string to the specified type, using for example, functions such as ***atoi()*** or ***atof()***. Thus, the following construct can be used to obtain the value of a floating point argument named "f":

```
float f = args["f"].AsDouble();
```

An exception will be thrown with an appropriate error message, if:

- the conversion fails, or
- "f" was described as an optional key or positional argument without default value (i.e. using ***AddOptional***()*** method), and it was not defined in the command line. Note that you can check for this case using the ***CArgValue::HasValue()*** method.

Code Examples

A simple application program, *test_ncbiargs_sample.cpp* demonstrates the usage of these classes for argument processing. See also *test_ncbiargs.cpp* (especially ***main()***, ***s_InitTest0()*** and ***s_RunTest0()*** there), and *asn2asn.cpp* for more examples.

Namespace, Name Concatenation and Compiler Specific Macros

The *ncbistl.hpp* provides a number of macros on namespace usage, name concatenation and macros for handling compiler specific behavior.

These topics are discussed in greater detail in the following subsections

- NCBI Namespace
- Other Name Space Macros
- Name Concatenation
- Compiler Specific Macros

NCBI Namespace

All new NCBI classes must be in the `ncbi::` namespace to avoid naming conflicts with other libraries or code. Rather than enclose all newly defined code in the following it is, from a stylistic point of view, better to use specially defined macros such as `BEGIN_NCBI_SCOPE`, `END_NCBI_SCOPE`, `USING_NCBI_SCOPE`:

```
namespace ncbi {
    // Indented code etc.

}
```

The use of `BEGIN_NCBI_SCOPE`, `END_NCBI_SCOPE`, `USING_NCBI_SCOPE` is discussed in use of the NCBI name scope.

Other Name Space Macros

The `BEGIN_NCBI_SCOPE`, `END_NCBI_SCOPE`, `USING_NCBI_SCOPE` macros in turn use the more general purpose `BEGIN_SCOPE(ns)`, `END_SCOPE(ns)`, `USING_SCOPE(ns)` macros, where the macro parameter `ns` is the namespace being defined. All NCBI related code should be in the `ncbi::` namespace so the `BEGIN_NCBI_SCOPE`, `END_NCBI_SCOPE`, `USING_NCBI_SCOPE` should be adequate for new NCBI code. However, in those rare circumstances, if you need to define a new name scope, you can directly use the `BEGIN_SCOPE(ns)`, `END_SCOPE(ns)`, `USING_SCOPE(ns)` macros.

Name Concatenation

The macros `NCBI_NAME2` and `NCBI_NAME3` define concatenation of two and three names respectively. These are used to build names for program generated class, struct, or method names.

Compiler Specific Macros

To cater to the idiosyncracies of compilers that have non-standard behavior, certain macros are defined to normalize their behavior.

The BREAK(it) macro advances the iterator to end of the loop and then breaks out of loop for the Sun WorkShop compiler with version less than 5.3. This is done because this compiler fails to call destructors for objects created in for-loop initializers. This macro prevents trouble with iterators that contain CRefs by advancing them to the end using a while-loop, thus avoiding the "deletion of referenced CObject" errors. For other compilers BREAK(it) is defined as the keyword break.

The ICC compiler may fail to generate code preceded by template<>. So use the macro EMPTY_TEMPLATE instead which expands to an empty string for the ICC compiler and to template<> for all other compilers.

For MSVC v6.0, the for keyword is defined as a macro to overcome a problem with for-loops in the compiler. The local variables in a for-loop initalization are visible outside the loop:

```
for (int i; i < 10; ++i) {
    // scope of i
}

// i should not be visible, but is visible in MSVC 6.0
```

Another macro called NCBI_EAT_SEMICOLON is used in creating new names which can allow a tailing semicolon without producing a compiler warning in some compilers.

Using the CNcbiRegistry Class

This section provides reference information on the use of the CNcbiRegistry class. For an overview of these classes refer to the introductory chapter.

The following topics are discussed in this section:

- Working with the Registry class: CNcbiRegistry
- Syntax of the Registry Configuration File
- Search Order for Initialization (*.ini) Files
- Setting Persistency and Modifiability of Registry Parameters Using CNcbiRegistry::EFlags
- Main Methods of CNcbiRegistry
- Additional Registry Methods

Working with the Registry class: CNcbiRegistry

The ***CNcbiRegistry*** class is used to access, modify and store runtime information read from a configuration file. The registry classes can be used to perform operations such as reading and parsing configuration files, searching, and editing the retrieved configuration information, and writing information back to configuration file.

If you are programming in the standard C++ Toolkit framework, using ***CNcbiApplication*** class, you should definitely read the rules on where the application is looking for the configuration file, and how you can access the loaded application-wide coniguration (registry) from your code. This is described in the earlier discussion on the CNcbiRegistry class.

Syntax of the Registry Configuration File

The configuration file is composed of *section* headers and "*name*=*value*" strings, which occur within the named sections. It is also possible to include comments in the file, which are indicated by a new line with a leading semi-colon. An example configuration file is shown below.

An Example Configuration File

```
# Registry file comment (begin of file)
# MyProgram.ini

; parameters for section1
[section1]
name1 = value1 and value1.2
n-2.3 = "  this value has two spaces at its very beginning and at the
end  "
name3 = this is a multi\
line value
name4 = this is a single line ended by back slash\\
name5 = all backslashes and \
new lines must be \\escaped\\...

[ section2.9-bis ]
; This is a comment...
name2 = value2
```

All comments and empty lines are ignored by the registry file parser. Line continuations, as usual, are indicated with a backslash escape. More generally, backslashes are processed as:

- [backslash] + [backslash] -- converted into a single [backslash]
- [backslash] + [space(s)] + [EndOfLine] -- converted to an [EndOfLine]
- [backslash] + ["] -- converted into a ["]

Character strings with embedded spaces do not need to be quoted, and an unescaped double quote at the very beginning or end of a value is ignored. All other combinations with [backslash] and ["] are invalid.

The following restrictions apply to the *section* and *name* identifiers occurring in a registry file:

- the string must contain only: [a-z], [A-Z], [0-9], [_.-] characters

- the interpretation of the string is **not** case-sensitive, e.g. *PATH* == *path* == *PaTh*
- all leading and trailing spaces will be truncated

Search Order for Initialization (*.ini) Files

On Unix platforms if an application *dir1/app1* is a symlink to *dir2/app2*, the search order for initialization files (*.ini files) will be as shown below:

1. *./app1.ini*
2. *$NCBI/app1.ini*
3. *~/app1.ini*
4. *dir1/app1.ini*
5. *dir2/app1.ini*
6. *./app2.ini*
7. *$NCBI/app2.ini*
8. *~/app2.ini*
9. *dir1/app2.ini*
10. *dir2/app2.ini*

Setting Persistency and Modifiability of Registry Parameters Using CNcbiRegistry::EFlags

In addition to the constructor, which initializes the registry in memory and loads parameters from a file, the ***CNcbiRegistry*** class provides methods for saving the registry (as a new configuration file), and reading in additonal parameters from a secondary file(s). Each "*name*=*value*" pair stored in the registry has three attributes, specifying whether or not that value is:

- *persistent*, meaning the value will be written to a file when the registry is saved
- *overridable* meaning the value can be overridden by a new value with the same name
- *truncatable* meaning that leading and trailing blanks can be truncated from the value

By default, all of the configuration's parameters are *persistent, overridable*, and *truncatable*. The ***EFlags*** enumeration that quantifies these attributes, and the related *typedef***TFlags**, are defined as:

```
/// Registry parameter settings.
///
```

```
/// A Registry parameter can be either transient or persistent,
/// overrideable or not overridable, truncatable or not truncatable.
enum EFlags {
    eTransient   = 0x1,             ///< Transient -- Wont be saved
    ePersistent  = 0x100,         ///< Persistent -- Saved when file is written
    eOverride    = 0x2,             ///< Existing value can be overriden
    eNoOverride  = 0x200,          ///< Cannot change existing value
    eTruncate   = 0x4,             ///< Leading, trailing blanks can be truncated
    eNoTruncate  = 0x400             ///< Cannot truncate parameter value
};
typedef int TFlags;  ///< Binary OR of "EFlags"
```

TFlags is simply used to clarify that an ***int*** derived from a combination (bit-wise *OR*) of ***EFlags*** is expected - not just an arbitrary "regular" ***int***. Many of ***CNcbiRegistry***'s methods take an optional ***TFlags*** argument, which qualifies the selected values with respect to these attributes.

For example, the following code excerpt sets the value of registry entry *MyName* in section *MySection* to *"Eugene"*. In particular, the ***TFlags*** argument, derived from the bit-wise OR of `eTruncate` and `eNoOverride`, specifies that (1) all leading and trailing blanks in the new value should be truncated, and (2) that the new value **cannot** be applied to override a previous value if one exists:

```
CNcbiRegistry reg(.....);
...
reg.Set("MySection", "MyName", " Eugene   ",
        CNcbiRegistry::eNoOverride | CNcbiRegistry::eTruncate);
```

Main Methods of *CNcbiRegistry*

The ***CNcbiRegistry*** class constructor takes two arguments - an input stream to read the registry from (usually a file), and an optional ***TFlags*** argument, where the latter can be used to specify that all of the values should be stored as *transient* rather than in the default mode, which is *persistent*:

```
CNcbiRegistry(CNcbiIstream& is, TFlags flags = 0);
```

Once the registry has been initialized by its constructor, it is also possible to load additional parameters from other file(s) using the ***Read()*** method:

```
void Read(CNcbiIstream& is, TFlags flags = 0);
```

Valid flags for the ***Read()*** method include `eTransient` and `eNoOverride`. The default is for all values to be read in as *persistent*, with the capability of overriding any previously loaded value associated with the same name. Either or both of these defaults can be modified by specifying `eTransient, eNoOverride`, or (`eTransient | eNoOverride`) as the `flags` argument in the above expression.

The ***Write()*** method takes as its sole argument, a destination stream to which only the *persistent* configuration parameters will be written.

```
bool Write(CNcbiOstream& os) const;
```

The configuration parameter values can also be set directly inside your application, using:

```
bool Set(const string& section, const string& name, const string& value, TFlags flags =
0);
```

Here, valid flag values include `ePersistent, eNoOverride`, `eTruncate,` or any logical combination of these. If `eNoOverride` is set and there is a previously defined value for this parameter, then the value is not reset, and the method returns *false*.

The ***Get()*** method first searches the set of *transient* parameters for a parameter named `name`, in section `section`, and if this fails, continues by searching the set of *persistent* parameters. However, if the `ePersistent` flag is used, then only the set of *persistent* parameters will be searched. On success, ***Get()*** returns the stored value. On failure, the empty string is returned.

```
const string& Get(const string& section, const string& name, TFlags flags = 0) const;
```

Additional Registry Methods

Four additional note-worthy methods defined in the ***CNcbiRegistry*** interface are:

```
bool Empty(void) const;
void Clear(void);
void EnumerateSections(list<string>*sections) const;
void EnumerateEntries(const string& section,  list<string>* entries) const;
```

Empty() returns *true* if the registry is empty. ***Clear()*** empties out the registry, discarding all stored parameters. ***EnumerateSections()*** writes all registry section names to the list of strings parameter named "sections". ***EnumerateEntries()*** writes the list of parameter names in section to the list of strings parameter named "entries".

Portable Stream Wrappers

Because of differences in the C++ standard stream implementations between different compilers and platforms, the file ***ncbistre.hpp*** contains portable aliases for the standard classes. To provide portability between the supported platforms, it is is recommended the definitions in ***ncbistre.hpp*** be used.

The ***ncbistre.hpp*** defines wrappers for many of the standard stream classes and contains conditional compilation statements triggered by macros to include portable definitions. For example, not all compilers support the newer '#include <iostream>' form. In this case, the older '#include <iostream.h>' is used based on whether the macro `NCBI_USE_OLD_IOSTREAM` is defined.

Instead of using the iostream, istream or ostream, you should use the portable ***CNcbiIostream***, ***CNcbiIstream*** and ***CNcbiOstream***. Similarly, instead of using the standard `cin`, `cout`, `cerr` you can use the more portable `NcbiCin`, `NcbiCout`, and `NcbiCerr`.

The ***ncbistre.hpp*** also defines functions that handle platform-specific end of line reads. For example, ***Endl()*** represents platform specific end of line, and ***NcbiGetline()*** reads from a specified input stream to a string, and ***NcbiGetlineEOL()*** reads from a specified input stream to a string taking into account platform specific end of line.

Working with Diagnostic Streams (*)

This section provides reference information on the use of the diagnostic stream classes. For an overview of the diagnostic stream concepts refer to the introductory chapter.

The ***CNcbiDiag*** class implements the functionality of an output stream enhanced with error posting mechanisms similar to those found in the NCBI C Toolkit. A ***CNcbiDiag*** object has the look and feel of an output stream; its member functions and friends include output operators and format manipulators. A ***CNcbiDiag*** object is not itself a stream, but serves as an interface to a stream which allows multiple threads to write to the same output. Each instance of ***CNcbiDiag*** includes the following private data members:

- a buffer to store (a single) message text
- a severity level
- a set of post flags

Limiting each instance of ***CNcbiDiag*** to the storage and handling of a single message ensures that multiple threads writing to the same stream will not have interleaving message texts.

The following topics are discussed in this section:

- Setting Diagnostic Severity Levels
- Controlling Apperance of Diagnostic Message using Post Flags
- Defining the Output Stream
- The Message Buffer
- Error codes and their Descriptions
- Defining Custom Handlers using CDiagHandler
- The ERR_POST Macro
- The _TRACE macro
- Example Usage of the CNcbiDiag class

Setting Diagnostic Severity Levels

Each ***CNcbiDiag*** instance has its own (***EDiagSev***) severity level, which is compared to a global severity threshold to determine whether or not its message should be posted. Six levels of severity are defined by the ***EDiagSev*** enumeration:

```
/// Severity level for the posted diagnostics.
enum EDiagSev {
    eDiag_Info = 0, ///< Informational message
    eDiag_Warning,  ///< Warning message
    eDiag_Error,    ///< Error message
    eDiag_Critical, ///< Critical error message
    eDiag_Fatal,    ///< Fatal error -- guarantees exit(or abort)
    eDiag_Trace,    ///< Trace message
    // Limits
    eDiagSevMin = eDiag_Info,  ///< Verbosity level for min. severity
    eDiagSevMax = eDiag_Trace  ///< Verbosity level for max. severity
};
```

The default is to post only those messages whose severity level exceeds the `eDiag_Warning` level (i.e. `eDiag_Error, eDiag_Critical`, and `eDiag_Fatal`). The global severity threshold for posting messages can be reset using ***SetDiagPostLevel (EDiagSev postSev)***. A parallel function, ***SetDiagDieLevel (EDiagSev dieSev)***, defines the severity level at which execution will abort.

Tracing is considered to be a special, debug-oriented feature, and therefore messages with severity level `eDiag_Trace` are not affected by these global *post/die* levels. Instead, ***SetDiagTrace (EDiagTrace enable, EDiagTrace default)*** is used to turn tracing on or off. By default, the tracing is off -- unless you assign the environment variable `$DIAG_TRACE` to an arbitrary non-empty string or, alternatively, define a *DIAG_TRACE* entry in the *[DEBUG]* section of your registry file.

The ***CNcbiDiag*** class also has class-specific *manipulators* to control the message severity level. These can be invoked as in the following examples on diagnostic stream `diag`:

```
diag << Info;    // set severity level to eDiag_Info
diag << Warning; // set severity level to eDiag_Warning
diag << Error;   // set severity level to eDiag_Error [default]
diag << Fatal;   // set severity level to eDiag_Fatal
diag << Trace;   // set severity level to eDiag_Trace
```

Controlling Apperance of Diagnostic Message using Post Flags

The post flags define additional information that will inserted into the output messages and appear along with the message body. The standard format of a message is:

```
"<file>", line <line>: <severity>: (<err_code>.<err_subcode>)
[<prefix1>::<prefix2>::<prefixN>] <message>\n
<err_code_message\n
<err_code_explanation>
```

where the each field are displayed (or not) depending on the post flags EDiagPostFlag associated with the ***CNcbiDiag***:

```
enum EDiagPostFlag {
    eDPF_File              = 0x1, ///< Set by default #if _DEBUG; else not set
    eDPF_LongFilename      = 0x2, ///< Set by default #if _DEBUG; else not set
    eDPF_Line              = 0x4, ///< Set by default #if _DEBUG; else not set
    eDPF_Prefix            = 0x8, ///< Set by default (always)
    eDPF_Severity          = 0x10,  ///< Set by default (always)
    eDPF_ErrCode           = 0x20,  ///< Set by default (always)
    eDPF_ErrSubCode        = 0x40,  ///< Set by default (always)
    eDPF_ErrCodeMessage    = 0x100, ///< Set by default (always)
    eDPF_ErrCodeExplanation = 0x200, ///< Set by default (always)
    eDPF_ErrCodeUseSeverity = 0x400, ///< Set by default (always)
    eDPF_DateTime          = 0x80,  ///< Include date and time
    eDPF_OmitInfoSev       = 0x4000,///< No severity indication if eDiag_Info

    /// Set all flags.
    eDPF_All               = 0x3FFF,

    /// Set all flags for using with __FILE__ and __LINE__.
    eDPF_Trace             = 0x1F,

    /// Print the posted message only;  without severity, location, prefix, etc.
    eDPF_Log               = 0x0,

    /// Ignore all other flags, use global flags.
    eDPF_Default           = 0x8000
};
```

The default message format displays only the severity level and the message body. This can be overridden inside the constructor for a specific instance of ***CNcbiDiag***, or globally, using ***SetDiagPostFlag(EDiagPostFlagflag)*** on a selected flag.

Defining the Output Stream

All ***CNcbiDiag*** objects are associated with a global output stream. The default is to post messages to `cerr` ouput stream, but the stream destination can be reset at any time using:

SetDiagStream(CNcbiOstream* os, bool quick_flush, FDiagCleanupcleanup, void* cleanup_data)

This function can be called numerous times, thus allowing different sections of the executable to write to different files. At any given time however, all ***CNcbiDiag*** objects will be associated with the same global output stream. Because the messages are completely buffered, each message will appear on whatever stream is active at the time the message actually completes.

And, of course, you can provide (using SetDiagHandler) your own message posting handler CDiagHandler, which does not necessarily write the messages to a standard C++ output stream. To preserve compatibility with old code, SetDiagHandler also continues to accept raw callback functions of type FDiagHandler.

The Message Buffer

The ***CNcbiDiag*** message buffer is initialized when the class is first instantiated. Additional information can then be appended to the message using the overloaded stream operator <<. Messages can then be terminated explicitly using CNcbiDiag's stream manipulator ***Endm***, or implicitly, when the ***CNcbiDiag*** object exits scope.

Implicit message termination also occurs as a side effect of applying one of the severity level manipulators. Whenever the severity level is changed, ***CNcbiDiag*** also automatically executes the following two *manipulators*:

- ***Endm*** -- the message is complete and the message buffer will be flushed
- ***Reset*** -- empty the contents of the current message buffer

When the message controlled by an instance of ***CNcbiDiag*** is complete, ***CNcbiDiag*** calls a global callback function (of type ***FDiagHandler***) and passes the message (along with its severity level) as the function arguments. The default callback function posts errors to the currently designated output stream, with the action (continue or abort) determined by the severity level of the message.

Error codes and their Descriptions

The ***CNcbiDiag*** class is capable of posting messages with error codes using the ErrCode*manipulator*. For example:

```
diag << ErrCode(2,1); // set error code 2, subcode 1
```

Error codes and subcodes are posted to an output stream only if applicable post flags were set. In addition to error codes, ***CNcbiDiag*** can also post their text explanations. It uses ***CDiagErrCodeInfo*** class to find an error message, which corresponds to a given error code/subcode. Such descriptions could be specified directly in the program code or placed in a separate message file. It is even possible to use several such files simultaneously. ***CDiagErrCodeInfo*** can also read error descriptions from any input stream(s), not necessarily files.

The following additional topics are discussed in the folllowing subsections:

- Preparing an Error Message File
- Using Error Codes in a Program

Preparing an Error Message File

The error message file contains plain ASCII text data. We would suggest using the *.msg* extension, but this is not mandatory. For example, the message file for an application named `SomeApp` might be called *SomeApp.msg*.

The message file must contain a line with the keyword *MODULE* in it, followed by the name of the module (in our example *SomeApp*). This line must be placed in the beginning of the file, before any other declarations. Lines with symbol # in the first position are treated as comments and ignored.

Here is an example of the message file:

```
# This is a message file for application "SomeApp"

MODULE SomeApp

# ------ Code 1 ------
$$ NoMemory, 1, Fatal : Memory allocation error

# ------ Code 2 ------
$$ File, 2, Critical : File error

$^ Open, 1 : Error open a specified file
This often indicates that the file simply does not exist.
Or, it may exist but you do not have permission to access the file in the requested mode.

$^ Read, 2, Error : Error read file
Not sure what would cause this...

$^ Write, 3, Critical
This may indicate that the filesystem is full.

# ------ Code 3 ------
$$ Math, 3
$^ Param, 20
$^ Range, 3
```

Lines beginning with *$$* define a top-level error code. Similarly, lines beginning with *$^* define subcodes of the top-level error code. In the above example *Open* is a subcode of *File* top-level error, which means the error with code 2 and subcode 1.

Both types of lines have similar structure:

```
$$/$^ <mnemonic_name>, <code> [, <severity> ] [: <message> ] \n
[ <explanation> ]
```

where

- `mnemonic_name` (*required*) Internal name of the error code/subcode. This is used as a part of an error name in a program code - so, it should also be a correct C/C++ identifier.
- `code` (*required*) Integer identifier of the error.

- `severity` (*optional*) This may be supplied to specify the severity level of the error. It may be specified as a severity level string (valid values are *Info, Warning, Error, Critical, Fatal, Trace*) or as an integer in the range from 0 (`eDiag_Info`) to 5 (`eDiag_Trace`). While integer values are acceptable, string values are more readable. If the severity level was not specified or could not be recognized, it is ignored, or inherited from a higher level (the severity of a subcode becomes the same as the severity of a top-level error code, which contains this subcode). As long as diagnostic `eDPF_ErrCodeUseSeverity` flag is set, the severity level specified in the message file overrides the one specified in a program, which allows for runtime customization. In the above example, *Critical* severity level will be used for all *File* errors, except *Read* subcode, which would have *Error* severity level.

- `message` (*optional*) Short description of the error. It must be a single-line message. As long as diagnostic `eDPF_ErrCodeMessage` flag is set, this message is posted as a part of the diagnostic output.

- `explanation` (*optional*) Following a top-level error code or a subcode definition string, it may be one or several lines of an explanation text. Its purpose is to provide additional information, which could be more detailed description of the error, or possible reasons of the problem. This text is posted in a diagnostic channel only if `eDPF_ErrCodeExplanaton` flag was set.

Using Error Codes in a Program

Taking a message file as an input, script ***msg2hpp.sh*** could generate a C/C++ header file with macro definitions of error codes. Based on our example, this script would generate the following:

```
#ifndef __MODULE_SomeApp__
#define __MODULE_SomeApp__

#define ERR_NoMemory 1,0
#define ERR_Fil 2,0
#define ERR_File_Open 2,1
#define ERR_File_Read 2,2
#define ERR_File_Write 2,3
#define ERR_Math 3,0
#define ERR_Math_Param 3,20
#define ERR_Math_Range 3,3

#endif
```

Having included this file in an application, it is possible to use mnemonic error names:

```
diag << ErrCode(ERR_File_Open);
```

instead of their numeric representations:

```
diag << ErrCode(2,1);
```

Defining Custom Handlers using CDiagHandler

The user can install his own handler (of type ***CDiagHandler***,) using ***SetDiagHandler()***. CDiagHandler is a simple abstract class:

```
class  CDiagHandler
{
public:
    /// Destructor.
    virtual ~CDiagHandler(void) {}

    /// Post message to handler.
    virtual void Post(const SDiagMessage& mess) = 0;
};
```

where ***SDiagMessage*** is a simple struct defined in *ncbidiag.hpp* whose data members' values are obtained from the ***CNcbiDiag*** object. The transfer of data values occurs at the time that ***Post*** is invoked. See also the section on Message posting for a more technical discussion.

The ERR_POST Macro

An `ERR_POST(message)` macro is also available for routine error posting. This macro implicitly creates a temporary ***CNcbiDiag*** object and puts the passed "message" into it with a default severity of `eDiag_Error`. A severity level manipulator can be applied if desired, to modify the message's severity level. For example:

```
long lll = 345;
ERR_POST("My ERR_POST message, print long: " << lll);
```

would write to the diagnostic stream something like:

```
"somefile.cpp", line 111: Error: My ERR_POST message, print long: 345
```

while:

```
double ddd = 123.345;
ERR_POST(Warning << "...print double: " << ddd);
```

would write to the diagnostic stream something like:

```
"somefile.cpp", line 222: Warning: ...print double: 123.345
```

The _TRACE macro

The `_TRACE(message)` macro is a debugging tool that allows the user to insert trace statements that will only be posted if the code was compiled in debug mode, and provided that the tracing has been turned on. If `DIAG_TRACE` is defined as an enviroment variable, or as an entry in the [DEBUG] section of your configuration file (**.ini*), the initial state of tracing is *on*. By default, if no such variable or registry entry is defined, tracing is *off*. ***SetDiagTrace (EDiagTrace enable, EDiagTrace default)*** is used to turn tracing on/off.

Just like ERR_POST, the _TRACE macro takes a message, and the message will be posted only if tracing has been enabled. For example:

```
SetDiagTrace(eDT_Disable);
_TRACE("Testing the _TRACE macro");
SetDiagTrace(eDT_Enable);
_TRACE("Testing the _TRACE macro AGAIN");
```

Here, only the second trace message will be posted, as tracing is disabled when the first _TRACE( ) macro call is executed.

Example Usage of the *CNcbiDiag* class

Normally, one should use ERR_POST() and _TRACE() macro to post messages, and regulate the severity level by using severity level manipulators, like:

```
ERR_POST(Info << "A notice" << "Fooo");
ERR_POST(Critical << "Some critical error");
```

Examples in *diag.cpp* demonstrate the use of ERR_POST and _TRACE. ***CTestApp::Run()*** begins by testing the ERR_POST and _TRACE macros. Initially, tracing is enabled (from the registry file), so the first _TRACE message is posted. Tracing is then explicitly disabled, so the second _TRACE message is **not** posted.

Next, the global severity level for posting messages is set to the lowest level (eDiag_Info) so that all but the trace messages will be visible. Trace messages are still disabled by the explicit call to ***SetDiagTrace()***. Five instances of the ***CNcbiDiag*** class are then created, each with an associated file name, line number, severity level, and enumerated value for the post flags. A single message, Msg will be posted on all of the diagnostic streams.

myHandler() is then installed to replace the default message handler. The last two messages, which are created after the new handler has been installed, are handled by ***myHandler()***. The first of these is a trace message however, and because tracing is now disabled, this message will not be made visible. All of the messages which do not explicitly use the ***Endm*** manipulator are automatically terminated when ***Run()*** exits.

Output generated by *diag.cpp*:

```
"/home/zimmerma/internal/c++/src/Demos/DiagStream/diag.cpp",
   line 23: Error: My ERR_POST message, print long: 345
"/home/zimmerma/internal/c++/src/Demos/DiagStream/diag.cpp",
   line 26: Warning: ...print double: 123.345
"/home/zimmerma/internal/c++/src/Demos/DiagStream/diag.cpp",
   line 34: Trace: Testing the _TRACE macro
"diag.cpp", line 41: Info: This is a test message
"diag.cpp", line 42: Warning: This is a test message
"diag.cpp", line 43: Error: This is a test message
Installed Handler "diag.cpp", line 45: Critical: This is a test message
```

Debug Macros

A number of debug macros such as _TRACE, _TROUBLE, _ASSERT, _VERIFY, _DEBUG_ARG can be used when the _DEBUG macro is defined.

These macros are part of CORELIB. However, they are discussed in a separate chapter on Debugging, Exceptions, and Error Handling.

Handling Exceptions

The CORELIB defines an extended exception handling mechanism based on the C++ std:: exception, but which considerably extends this mechanism to provide a backlog, history of unfinished tasks, and more meaningful reporting on the exception itself.

While the extended exception handling mechanism is part of CORELIB, it is discussed in a separate chapter on Debugging, Exceptions, and Error Handling.

Defining the Standard NCBI C++ types and their Limits

The following section provides a reference to the files and limit values used to in the C++ Toolkit to write portable code. An introduction to the scope of some of these portability definitions is presented the introduction chapter.

The following topics are discussed in this section:

- Headers Files containing Portability Definitions
- Built-in Integral Types
- Auxiliary Types
- Fixed-size Integer Types
- The "Ncbi_BigScalar" Type
- Encouraged and Discouraged Types

Headers Files containing Portability Definitions

- *corelib/ncbitype.h* -- definitions of NCBI fixed-size integer types
- *corelib/ncbi_limits.h* -- numeric limits for:
    - NCBI fixed-size integer types
    - built-in integer types
    - built-in floating-point types

- *corelib/ncbi_limits.hpp* -- temporary (and incomplete) replacement for the Standard C++ Template Library's API

Built-in Integral Types

We encourage the use of standard C/C++ types shown in Table 4, and we state that the following assumptions (no less, no more) on their sizes and limits will be valid for all supported platforms:

Table 4. Standard C/C++ Types

Name	Size(bytes)	Min	Max	Note
char	1	`kMin_Char` (0 or -128)	`kMax_Char` (256 or 127)	It can be either signed or unsigned! Use it wherever you dont care of +/- (e.g. in character strings).
signed char	1	`kMin_SChar` (-128)	`kMax_SChar` (127)	
unsigned char	1	`kMin_UChar` (0)	`kMax_UChar` (255)	
short, signed short	2 or more	`kMin_Short` (-32768 or less)	`kMax_Short` (32767 or greater)	Use ***"int"*** if size isn't critical
usigned short	2 or more	`kMin_UShort` (0)	`kMax_UShort` (65535 or greater)	Use ***"unsigned int"*** if size isn't critical
`int`, `signed int`	4 or more	`kMin_Int` (-2147483648 or less)	`kMax_Int` (2147483647 or greater)	
unsigned int	4 or more	`kMin_UInt` (0)	`kMax_UInt` (4294967295 or greater)	
double	4 or more	`kMin_Double`	`kMax_Double`	

Types "long" and "float" are discouraged to use in the portable code.
Type "long long" is prohibited to use in the portable code.

Auxiliary Types

Use type ***"bool"*** to represent boolean values. It accepts one of { *false*, *true* }.

Use type ***"size_t"*** to represent a size of memory structure, e.g. obtained as a result of *sizeof* operation.

Use type ***"SIZE_TYPE"*** to represent a size of standard C++ ***"string"***, -- this is a portable substitution for ***"std::string::size_type"***.

Fixed-size Integer Types

Sometimes it is necessary to use an integer type which:

- has a well-known fixed size(and lower/upper limits)
- be just the same on all platforms(but maybe a byte/bit order, depending on the processor architecture)

NCBI C++ standard headers provide the fixed-size integer types shown in Table 5:

Table 5. Fixed-integer Types

Name	Size(bytes)	Min	Max
Char, Int1	1	`kMin_I1`	`kMax_I1`
Uchar, Uint1	1	0	`kMax_UI1`
Int2	2	`kMin_I2`	`kMax_I2`
Uint2	2	0	`kMax_UI2`
Int4	4	`kMin_I4`	`kMax_I4`
Uint4	4	0	`kMax_UI4`
Int8	8	`kMin_I8`	`kMax_I8`
Uint8	8	0	`kMax_UI8`

In Table 6, the `"kM*_*"` are constants of relevant fixed-size integer type. They are guaranteed to be equal to the appropriate *preprocessor constants* from the old NCBI C headers (`"INT*_M*"`). Please also note that the mentioned `"INT*_M*"` are not defined in the C++ headers -- in order to discourage their use in the C++ code.

Table 6. Correspondence between the kM*_* constants and the old style INT*_M* constants

Constant(NCBI C++)	Value	Define(NCBI C)
`kMin_I1`	-128	`INT1_MIN`
`kMax_I1`	+127	`INT1_MAX`
`kMax_UI1`	+255	`UINT1_MAX`
`kMin_I2`	-32768	`INT2_MIN`
`kMax_I2`	+32767	`INT2_MAX`
`kMax_UI2`	+65535	`UINT2_MAX`
`kMin_I4`	-2147483648	`INT4_MIN`
`kMax_I4`	+2147483647	`INT4_MAX`
`kMax_UI4`	+4294967295	`UINT4_MAX`
`kMin_I8`	-9223372036854775808	`INT8_MIN`
`kMax_I8`	+9223372036854775807	`INT8_MAX`
`kMax_UI8`	+18446744073709551615	`UINT8_MAX`

The *"Ncbi_BigScalar"* Type

NCBI C++ standard headers also define a special type ***"Ncbi_BigScalar"***. The only assumption that can be made(and used in your code) is that ***"Ncbi_BigScalar"*** variable has a size which is enough to hold any integral, floating-point or pointer variable like ***"Int8"***, or ***"double"("long double")***, or ***"void*"***. This type can be useful e.g. to hold a callback data of arbitrary fundamental type; however, in general, the use of this type is discouraged.

Encouraged and Discouraged Types

For the sake of code portability and for better compatibility with the third-party and system libraries, one should follow the following set of rules:

- Use standard C/C++ integer types ***"char", "signed char", "unsigned char", "short", "unsigned short", "int", "unsigned int"*** in **any** case where the assumptions made for them in Table 4 are enough.
- It is not recommended to use ***"long"*** type unless it is absolutely necessary (e.g. in the lower-level or third-party code), and even if you have to, then try to localize the use of ***"long"*** as much as possible.
- The same(as for ***"long"***) is for the fixed-size types enlisted in Table 5. -- If you have to use these in your code, try to keep them inside your modules and avoid mixing them with standard C/C++ types (as in assignments, function arg-by-value passing and in arithmetic expressions) as much as possible.
- For the policy on other types see in sections "Auxiliary types" and "Floating point types".

Understanding Smart Pointers: the *CObject* and *CRef* Classes

This section provides reference information on the use of CRef and CObject classes. For an overview of these classes refer to the introductory chapter.

The following is a list of topics discussed in this section:

- STL auto_ptrs
- The CRef Class
- The CObject Class
- The CObjectFor class: using smart pointers for standard types
- When to use CRefs and auto_ptrs
- CRef Pitfalls

STL auto_ptrs

C programmers are well-acquainted with the advantages and pitfalls of using pointers. As is often the case, the good news is also the bad news:

- memory can be dynamically allocated as needed, but may not be deallocated as needed, due to unanticipated execution paths;
- void pointers allow heterogeneous function arguments of different types, but type information may not be there when you need it.

C++ adds some additional considerations to pointer management: STL containers cannot hold *reference* objects, so you are left with the choice of using either pointers or *copies* of objects. Neither choice is attractive, as pointers can cause memory leaks and the copy constructor may be expensive.

The idea behind a C++ *smart pointer* is to create a wrapper class capable of holding a pointer. The wrapper class's constructors and destructors can then handle memory management as the object goes in and out of scope. The problem with this solution is that it does not handle multiple pointers to the same resource properly, and it raises the issue of ownership. This is essentially what the ***auto_ptr*** offers, but this strategy is only safe to use when the resource maps to a single pointer variable.

For example, the following code has two very serious problems:

```
int* ip = new int(5);
{
    auto_ptr<int> a1(ip);
    auto_ptr<int> a2(ip);
}
*ip = 10/(*ip);
```

The first problem occurs inside the block where the two ***auto_ptr***s are defined. Both are referencing the same variable pointed to by yet another C pointer, and each considers itself to be the owner of that reference. Thus, when the block is exited, the *delete[]* operation is executed twice for the same pointer.

Even if this first problem did not occur - for example if only one ***auto_ptr*** had been defined - the second problem occurs when we try to dereference `ip`. The *delete* operation occurring as the block exits has now reset `*ip` to *0*, so an attempt to divide by zero occurs.

The problem with using ***auto_ptr*** is that it provides semantics of strict ownership. When an ***auto_ptr*** is destructed, it deletes the object it points to, and therefore the object should not be pointed to simulateously by others. Two or more ***auto_ptr***s should not own the same object; that is, point to the same object. This can occur if two ***auto_ptr***s are initialized to the same object, as seen in the above example where auto pointers `a1` and `a2` are both initialized with `ip`. In using ***auto_ptr***, the programmer must ensure that situations similar to the above do not occur.

See also STL auto_ptrs for additional discussion on using ***auto_ptr***.

The CRef (%20) Class

These issues are addressed in the NCBI C++ Toolkit by using *reference-counted* smart pointers: a resource cannot be deallocated until **all** references to it have ceased to exist. The implementation of a smart pointer in the NCBI C++ Toolkit is actually divided between two classes: ***CRef*** and ***CObject***.

The ***CRef*** class essentially provides a pointer interface to a ***CObject***, while the ***CObject*** actually stores the data and maintains the reference count to it. The constructor used to create a new ***CRef*** pointing to a particular ***CObject*** automatically increments the object's reference count. Similarly, the ***CRef*** destructor automatically decrements the reference count. In both cases however, the modification of the reference count is implemented by a member function of the ***CObject***. The ***CRef*** class itself does not have direct access to the reference count and contains only a single data member -- its pointer to a ***CObject***. In addition to the ***CRef*** class's constructors and destructors, its interface to the ***CObject*** pointer includes access/mutate functions such as:

```
bool Empty()
bool NotEmpty()
CObject* GetPointer()
CObject& GetObject()
CObject* Release()
void Reset(CObject* newPtr)
void Reset(void)
operator bool()
bool operator!()
CRefBase& operator=(const CRefBase& ref)
```

Both the ***Release()*** and ***Reset()*** functions set the ***CRef*** object's `m_ptr` to 0, thus effectively removing the reference to its ***CObject***. There are important distinctions between these two functions however. The ***Release()*** method removes the reference without destroying the object, while the ***Reset()*** method may lead to the destruction of the object if there are no other references to it.

If the ***CObject***'s internal reference count is 1 at the time ***Release()*** is invoked, that reference count will be decremented to 0, and a pointer to the ***CObject*** is returned. The ***Release()*** method can throw two types of exceptions: (1) a *null pointer* exception if `m_ptr` is already *0*, and (2) an *Illegal CObject::ReleaseReference()* exception if there are currently other references to that object. An object must be free of all references (but this one) before it can be "released". In contrast, the ***Reset(void)*** function simply resets the ***CRef***'s `m_ptr` to 0, decrements the ***CObject***'s reference count, and, if the ***CObject*** has no other references and was dynamically allocated, deletes the ***CObject***.

Each member function of the ***CRef*** class also has a *const* implementation that is invoked when the pointer is to a *const* object. In addition, there is also a ***CConstRef*** class that parallels the ***CRef*** class. Both ***CRef*** and ***CConstRef*** are implemented as template classes, where the template argument specifies the type of object which will be pointed to. For example, in the section on Traversing an ASN.1 Data Structure we examined the structure of the ***CBiostruc*** class and found the following type definition

```
typedef list< CRef< ::CBiostruc_id > > TId;
```

As described there, this *typedef* defines ***TId*** to be a list of pointers to ***CBiostruc_id*** objects. And as you might expect, ***CBiostruc_id*** is a specialized subclass of ***CObject***.

The CObject (%20) Class

The ***CObject*** class serves as a base class for all objects requiring a reference count. There is little overhead entailed by deriving a new class from this base class, and most objects in the NCBI C++ Toolkit are derived from the ***CObject*** class. For example, ***CNCBINode*** is a direct descendant of ***CObject***, and all of the other `HTML` classes descend either directly or indirectly from ***CNCBINode***. Similarly, all of the ASN.1 classes defined in the *include/objects* directory, as well as many of the classes defined in the *include/serial* directory are derived either directly or indirectly from the ***CObject*** class.

The ***CObject*** class contains a single private data member, the reference counter, and a set of member functions which provide an interface to the reference counter. As such, it is truly a base class which has no stand-alone utility, as it does not even provide allocation for data values. It is the *descendant* classes, which inherit all the functionality of the ***CObject*** class, that provide the necessary richness in representation and allocation required for the widely diverse set of objects implemented in the NCBI C++ Toolkit. Nevertheless, it is often necessary to use smart pointers on simple data types, such as ***int***, ***string*** etc. The ***CObjectFor*** class, described below, was designed for this purpose.

The CObjectFor (%20) class: using smart pointers for standard types

The ***CObjectFor*** class is derived directly from ***CObject***, and is implemented as a template class whose argument specifies the standard type that will be pointed to. In addition to the reference counter inherited from its parent class, ***CObjectFor*** has a private data member of the parameterized type, and a member function ***GetData()*** to access it.

An example program, *smart.cpp*, uses the ***CRef*** and ***CObjectFor*** classes, and demonstrates the differences in memory management that arise using ***auto_ptr*** and ***CRef***. Using an ***auto_ptr*** to reference an ***int***, the program tests whether or not the reference is still accessible after an auxilliary ***auto_ptr*** which goes out of scope has also been used to reference it. The same sequence is then tested using ***CRef*** objects instead.

In the first case, the original auto_ptr, `orig_ap`, becomes *NULL* at the moment when ownership is transferred to `copy_ap` by the copy constructor. Using ***CRef*** objects however, the reference contained in the original ***CRef*** remains accessible (via `orig`) in all blocks where `orig` is defined. Moreover, the reference itself, i.e. the object pointed to, continues to exist until **all** references to it have been removed.

When to use CRefs and auto_ptrs

There is some overhead in using ***CRef*** and ***auto_ptr***, and these objects should only be used where needed. Memory leaks are generally caused as a result of unexpected execution paths. For example:

```
{
    int *num = new int(5);
    ComplexFunction (num);
    delete num;
    ...
}
```

If ***ComplexFunction()*** executes normally, control returns to the block where it was invoked, and memory is freed by the *delete* statement. Unforeseen events however, may trigger exceptions, causing control to pass elsewhere. In these cases, the *delete* statement may never be reached. The use of a ***CRef*** or an ***auto_ptr*** is appropriate for these situations, as they both guarantee that the object will be destroyed when the reference goes out of scope.

One situation where they may not be required is when a pointer is embedded inside another object. If that object's destructor also handles the deallocation of its embedded objects, then it is sufficient to use a ***CRef*** on the containing object only.

CRef Pitfalls

Inadvertent Object Destruction

When the last reference to a ***CRef*** object goes out of scope or the ***CRef*** is otherwise marked for garbage collection, the object to which the ***CRef*** points is also destroyed. This feature helps to prevent memory leaks, but it also requires care in the use of ***CRef***s within methods and functions.

```
class CMy : public CObject
{
.....
};

void f(CMy* a)
{
    CRef b = a;
    return;
}

CMy* a = new CMy();
f(a);
// the object "a" is now destroyed!
```

In this example the function ***f()*** establishes a local ***CRef*** to the ***CMy*** object a. On exitting ***f()*** the ***CRef***b is destroyed, including the implied destruction of the ***CMy*** objects a. To avoid this behavior, pass a ***CRef*** to the function ***f()*** instead of a normal pointer variable:

```
CRef a = new CMy();
f(a);
// the CMy object pointed to by "a" is not  destroyed!
```

Atomic Counters

The CORELIB implements efficient atomic counters that are used for ***CObject*** reference counts. The classes ***CAtomicCounter*** and ***CMutableAtomicCounter*** provide respectively a base atomic counter class, and a mutable atomic counter for multithreaded applications. These classes are used to in reference counted smart pointers.

The ***CAtomicCounter*** base class provides the base methods ***Get()***, ***Set()***, ***Add()*** for atomic counters:

```
class CAtomicCounter
{
public:
    typedef TNCBIAtomicValue TValue;  ///< Alias TValue for TNCBIAtomicValue

    /// Get atomic counter value.
    TValue Get(void) const THROWS_NONE;

    /// Set atomic counter value.
    void   Set(TValue new_value) THROWS_NONE;

    /// Atomically add value (=delta), and return new counter value.
    TValue Add(int delta) THROWS_NONE;

    /// Define NCBI_COUNTER_ADD if one has not been defined.
#if defined(NCBI_COUNTER_USE_ASM)
    static TValue x_Add(volatile TValue* value, int delta) THROWS_NONE;
#  if !defined(NCBI_COUNTER_ADD)
#     define NCBI_COUNTER_ADD(value, delta) NCBI_NS_NCBI::CAtomicCounter::x_Add((value),
(delta))
#  endif
#endif

private:
     ...

};
```

`TNCBIAtomicValue` is defined as a macro and its definition is platform dependent. If threads are not used (Macro `NCBI_NO_THREADS` defined), `TNCBIAtomicValue` is an ***unsigned int*** value. If threads are used, then a number of defines in file "*ncbictr.hpp*" ensure that a platform specific definition is selected for `TNCBIAtomicValue`.

The ***CMutableAtomicCounter*** uses the ***CAtomicCounter*** as its internal structure of the atomic counter but declares this counter value as mutable. The ***Get()***, ***Set()***, ***Add()*** methods for ***CMutableAtomicCounter*** are implemented by calls to the corresponding ***Get()***, ***Set()***, ***Add()*** methods for the ***CAtomicCounter***:

```
class  CMutableAtomicCounter
```

```
{
public:
    typedef CAtomicCounter::TValue TValue; ///< Alias TValue simplifies syntax

    /// Get atomic counter value.
    TValue Get(void) const THROWS_NONE
        { return m_Counter.Get(); }

    /// Set atomic counter value.
    void   Set(TValue new_value) const THROWS_NONE
        { m_Counter.Set(new_value); }

    /// Atomically add value (=delta), and return new counter value.
    TValue Add(int delta) const THROWS_NONE
        { return m_Counter.Add(delta); }

private:
    ...
};
```

Portable mechanisms for loading DLLs

The ***CDll*** class defines a portable way of dynamically loading shared libraries and finding entry points for functions in the libraray. Currently this portable behavior is defined for Unix and MS-Windows platforms only. On Unix systems loading of the shared library is implemented using the Unix system call ***dlopen()*** and the entry point address obtained using the Unix system call ***dlsym()***. On MS Windows systems the system call ***LoadLibraray()*** is used to load the library, and the system call ***GetProcAddress()*** is used to get a function's entry point address.

All methods of ***CDll*** class, except the destructor, throw the exception *CCoreException::eDll* on error.

You can specify when to load the DLL -- when the ***CDll*** object is created (loading in the constructor), or by an explicit call to ***CDll::Load()***. You can also specify whether the DLL is unloaded automatically when ***CDll's*** destructor is called or if the DLL should remain loaded in memory. This behavior is controlled by arguments to ***CDll's*** constructor.

The following additional topics are described in this section:

- CDll Constructor
- CDll Basename
- Other CDll Methods

CDll Constructor

The CDll constructor has two forms:

- Constructor 1:

```
CDll(const string& name,
      ELoad         when_to_load = eLoadNow,
      EAutoUnload   auto_unload  = eNoAutoUnload,
      EBasename     treate_as    = eBasename);
```

- Constructor 2:

```
 CDll(const string& path, const string& name,
      ELoad         when_to_load = eLoadNow,
      EAutoUnload   auto_unload  = eNoAutoUnload,
      EBasename     treate_as    = eBasename);
```

The two constuctor forms are very similar with the exception that constructor 2 uses two parameters: the `path` and `name` parameters to build a path to the DLL, whereas in constructor 1, the name parameter contains the full path to the DLL. The other parameters in the constructors are the same.

The parameter `when_to_load` is defined as an enum type of ***ELoad*** and has the values *eLoadNow* or *eLoadLater*. When *eLoadNow* is passed to the constructor (default value), the DLL is loaded in the constructor; otherwise it has to be loaded via an explicit call to the ***Load()*** method.

The parameter `auto_load` is defined as an enum type of ***EAutoLoad*** and has the values *eAutoUnload* or *eNoAutoUnload*. When *eAutoUnload* is passed to the constructor (default value), the DLL is unloaded in the destructor; otherwise it will remain loaded in memory.

The parameter `treat_as` is defined as an enum type of ***EBasename*** and has the values *eBasename* or *eExactName*. When *eBasename* is passed to the constructor (default value), the name parameter is treated as a basename if it looks like one; otherwise the exact name or "as is" value is used with no addition of prefix or suffix.

CDll Basename

The DLL name is considered the basename if it does not contain embedded '/', '\', or ':' symbols. Also, in this case, if the DLL name does not match the pattern "lib*.so", "lib*.so.*", or "*.dll" and if *eExactName* flag is not passed to the constructor, then it will be automagically transformed according to the following rules:

- UNIX: <name> -> lib<name>.so
- MS Windows: <name> -> <name>.dll

If the DLL is specified by its basename, then it will be searched after the transformation described above in the following locations:

- UNIX:

  1. The directories that are listed in the `LD_LIBRARY_PATH` environment variable which are analyzed once at the process startup.
  2. The directory from which the application loaded
  3. Hard-coded (e.g. with `ldconfig' on Linux) paths

- MS Windows:

  1. The directory from which the application is loaded
  2. The current directory
  3. The Windows system directory
  4. The Windows directory
  5. The directories that are listed in the `PATH` environment variable

Other CDll Methods

Two methods mentioned earlier for the ***CDll*** class are the ***Load()*** and ***Unload()*** methods. The ***Load()*** method loads the DLL using the name specified in the constructor's DLL `name` parameter. The ***Load()*** method is expected to be used when the DLL is not explictly loaded in the constructor. That is, when the ***CDll*** constructor is passed the *eLoadLater* parameter. If the ***Load()*** is called more than once without calling ***Unload()*** in between, then it will do nothing. The syntax of the ***Load()*** methods is

```
void Load(void);
```

The ***Unload()*** method unloads that DLL whose name was specified in the constructor's DLL `name` parameter. The ***Unload()*** method is expected to be used when the DLL is not explictly unloaded in the destructor. This occurs, when the ***CDll*** constructor is passed the *eNoAutoUnload* parameter. If the ***Unload()*** is called when the DLL is not loaded, then it will do nothing. The syntax of the ***Unload()*** methods is

```
void Unload(void);
```

Once the DLL is loaded, you can call the DLL's functions by first getting the function's entry point (address), and using this to call the function. The function template ***GetEntryPoint()*** method is used to get the entry point address and is defined as:

```
template <class TPointer>
TPointer GetEntryPoint(const string& name, TPointer* entry_ptr);
```

This method returns the entry point's address on success, or NULL on error. If the DLL is not loaded when this method is called, then this method will call ***Load()*** to load the DLL which can result in throwing an exception if ***Load()*** fails.

Some sample code illustrating the use of these methods is shown in *src/corelib/test/test_ncbidll.cpp*

Executing Commands and Spawing Processes using the CExec class

The ***CExec*** defines a portable execute class that can be used to execute system commands and spawn new processes.

The following topics relating to the ***CExec*** class are discussed, next:

- Executing a System Command using the System() Method
- Defining Spawned Process Modes (EMode type)
- Spawning a Process using SpawnX() Methods
- Waiting for a Process to Terminate using the Wait() method

Executing a System Command using the System() Method

You can use the class-wide ***CExec::System()*** method to execute a system command:

```
static int System(const char* cmdline);
```

CExec::System() returns the executed command's exit code and throws an exception if the command failed to execute. If cmdline is a null pointer, ***CExec::System()*** checks if the shell (command interpreter) exists and is executable. If the shell is available, ***System()*** returns a non-zero value; otherwise, it returns 0.

Defining Spawned Process Modes (EMode type)

The spawned process can be created in several modes defined by the enum type ***EMode*** . The meanings of the enum values for ***EMode*** type are:

- *eOverlay*: This mode overlays the calling process with new process, destroying the calling process.
- *eWait*: This mode suspends the calling thread until execution of a new process is complete. That is, the called process is called synchronously.
- *eNoWait*: This is the opposite of *eWait*. This mode continues to execute the calling process concurrently with the new called process in an asynchronous fashion.

- *eDetach*: This mode continues to execute the calling process and new process is "detached" and run in background with no access to console or keyboard. Calls to ***Wait()*** against new process will fail. This is an asynchronous spawn.

Spawning a Process using SpawnX() Methods

A new process can be spawned by calling any of the class-wide methods named ***SpawnX()*** which have the form:

```
static int SpawnX(const EMode mode, const char *cmdname, const char *argv, ... );
```

The parameter `mode` has the meanings discussed in the section Defining Spawned Process Modes (EMode type). The parameter `cmdname` is the command-line string to start the process, and parameter `argv` is the argument vector containing arguments to the process.

The ***X*** in the function name is a one to three letter suffix indicating the type of the spawn function. Each of the letters in the suffix ***X***, for ***SpawnX()*** has the following meanings:

- *L*: The letter "L" as suffix refers to the fact that command-line arguments are passed separately as arguments.
- *E*: The letter "E" as suffix refers to the fact that environment pointer, `envp`, is passed as an array of pointers to environment settings to the new process. The *NULL* environment pointer indicates that the new process will inherit the parents process's environment.
- *P*: The letter "P" as suffix refers to the fact that the `PATH` environment variable is used to find file to execute. Note that on a Unix platform this feature works in functions without letter "P" in the function name.
- *V*: The letter "V" as suffix refers to the fact that the number of command-line arguments are variable.

Using the above letter combinations as suffixes, the following spawn functions are defined:

- ***SpawnL()***: In the ***SpawnL()*** version, the command-line arguments are passed individually. ***SpawnL()*** is typically used when number of parameters to the new process is known in advance.
- ***SpawnLE()***: In the ***SpawnLE()*** version, the command-line arguments and environment pointer are passed individually. ***SpawnLE()*** is typically used when number of parameters to the new process and individual environment parameter settings are known in advance.
- ***SpawnLP()***: In the ***SpawnLP()*** version, the command-line arguments are passed individually and the `PATH` environment variable is used to find the file to execute. ***SpawnLP()*** is typically used when number of parameters to the new process is known in advance but the exact path to the executable is not known.

- ***SpawnLPE()***: In the ***SpawnLPE()*** the command-line arguments and environment pointer are passed individually, and the PATH environment variable is used to find the file to execute. ***SpawnLPE()*** is typically used when the number of parameters to the new process and individual environment parameter settings are known in advance, but the exact path to the executable is not known.
- ***SpawnV()***: In the ***SpawnV()*** version, the command-line arguments are a variable number. The array of pointers to arguments must have a length of 1 or more and you must assign parameters for the new process beginning from 1.
- ***SpawnVE()***: In the ***SpawnVE()*** version, the command-line arguments are a variable number. The array of pointers to arguments must have a length of 1 or more and you must assign parameters for the new process beginning from 1. The individual environment parameter settings are known in advance and passed explicitly.
- ***SpawnVP()***: In the ***SpawnVP()*** version, the command-line arguments are a variable number. The array of pointers to arguments must have a length of 1 or more and you must assign parameters for the new process beginning from 1. The PATH environment variable is used to find the file to execute.
- ***SpawnVPE()***: In the ***SpawnVPE()*** version, the command-line arguments are a variable number. The array of pointers to arguments must have a length of 1 or more and you must assign parameters for the new process beginning from 1. The PATH environment variable is used to find the file to execute, and the environment is passed via an environment vector pointer.

Refer to the *include/corelib/ncbiexec.hpp* file to view the exact form of the ***SpawnX()*** function calls.

Some sample code illustrating the use of these methods is shown in *src/corelib/test/test_ncbiexec.cpp*

Waiting for a Process to Terminate using the Wait() method

The ***CExec*** class defines a ***Wait()*** method that causes a process to wait until the child process terminates:

```
static int Wait(const int pid);
```

The argument to the ***Wait()*** method is the pid (process ID) of the child process on which the caller is waiting to terminate. ***Wait()*** returns immeditately if the specifed child process has already terminated and returns an exit code of the child process, if there are no errors; or a -1, if an error has occurred.

Implementing Parralelism using Threads and Synchronization Mechanisms

This section provides reference information on how to add multithreading to your application and how to use basic synchronization objects. For an overview of these concepts refer to the introductory topic on this subject.

Note that all classes are defined in *include/corelib/ncbithr.hpp* and *include/corelib/ncbimtx.hpp*.

The following topics are discussed in this section:

- Using Threads
- CThread class public methods
- CThread class protected methods
- Thread Life Cycle
- Referencing thread objects
- Synchronization
- Thread local storage (CTls<> class [*])

Using Threads

CThread class is defined in *include/corelib/ncbithr.hpp*. The ***CThread*** class provides all basic thread functionality: thread creation, launching, termination, and cleanup. To create user-defined thread one needs only to provide the thread's ***Main()*** function and, in some cases, create a new constructor to transfer data to the thread object, and override ***OnExit()*** method for thread-specific data cleanup. To create a custom thread:

1. Derive your class from ***CThread***, override ***Main()*** and, if necessary, ***OnExit()*** methods.
2. Create thread object in your application. You can do this only with *new* operator, since static or in-stack thread objects are prohibited (see below). The best way to reference thread objects is to use ***CRef<CThread>*** class.
3. Call ***Run()*** to start the thread execution.
4. Call ***Detach()*** to let the thread run independently (it will destroy itself on termination then), or use ***Join()*** to wait for the thread termination.

The code should look like:

```
#include <corelib/ncbistd.hpp>
#include <corelib/ncbithr.hpp>

USING_NCBI_SCOPE;
```

```
class CMyThread : public CThread
{
public:
    CMyThread(int index) : m_Index(index) {}
    virtual void* Main(void);
    virtual void OnExit(void);
private:
    int m_Index;
    int* heap_var;
};


void* CMyThread::Main(void)
{
    cout << "Thread " << m_Index << endl;

    heap_var = new int; // to be destroyed by OnExit()
    *heap_var = 12345;

    int* return_value = new int; // return to the main thread
    *return_value = m_Index;
    return return_value;
}

void CMyThread::OnExit(void)
{
    delete heap_var;
}

int main(void)
{
    CMyThread* thread = new CMyThread(33);
    thread->Run();
    int* result;
    thread->Join(reinterpret_cast<void**>(&result));
    cout << "Returned value: " << *result << endl;
    delete result;
    return 0;
}
```

The above simple application will start one child thread, passing *33* as the `index` value. The thread prints `"Thread 33"` message, allocates and initializes two integer variables, and terminates. The thread's ***Main()*** function returns a pointer to one of the allocated values. This pointer is then passed to ***Join()*** method and can be used by another thread. the other integer allocated by ***Main()*** is destroyed by ***OnExit()*** method.

It is important not to terminate the program until there are running threads. Program termination will cause all the running threads to terminate also. In the above example ***Join()*** function is used to wait for the child thread termination.

The following subsections discuss the individual classes in more detail.

CThread (%20) class public methods

CThread(void) Create the thread object (without running it). ***bool Run(void)*** Spawn the new thread, initialize internal ***CThread*** data and launch user-provided ***Main()***. The method guarantees that the new thread will start before it returns to the calling function. ***void Detach(void)*** Inform the thread that user does not need to wait for its termination. Detached thread will destroy itself after termination. If ***Detach()*** is called for a thread, which has already terminated, it will be scheduled for destruction immediately. Only one call to ***Detach()*** is allowed for each thread object. ***void Join(void** exit_data)*** Wait for the thread termination. ***Join()*** will store the ***void*** pointer as returned by the user's ***Main()*** method, or passed to the ***Exit()*** function to the `exit_data`. Then the thread will be scheduled for destruction. Only one call to ***Join()*** is allowed for each thread object. If called more than once, ***Join()*** will cause a runtime error. ***static void Exit(void* exit_data)*** This function may be called by a thread object itself to terminate the thread. The thread will be terminated and, if already detached, scheduled for destruction. `exit_data` value is transferred to the ***Join()*** function as if it was returned by the ***Main()***. ***Exit()*** will also call virtual method ***OnExit()*** to execute user-provided cleanup code (if any). ***bool Discard(void)*** Schedules the thread object for destruction if it has not been run yet. This function is provided since there is no other way to delete a thread object without running it. On success, return *true*. If the thread has already been run, ***Discard()*** do nothing and return *false*. ***static CThread::TID GetSelf(void)*** This method returns a unique thread ID. This ID may be then used to identify threads, for example, to track the owner of a shared resource. Since the main thread has no associated ***CThread*** object, a special value of 0 (zero) is reserved for the main thread ID.

CThread (%20) class protected methods

virtual void* Main(void)Main() is the thread's main function (just like an application ***main()*** function). This method is not defined in the ***CThread*** class. It must be provided by derived user-defined class. The return value is passed to the ***Join()*** function (and thus may be used by another thread for some sort of inter-thread communication). ***virtual void OnExit(void)*** This method is called (in the context of the thread) just before the thread termination to cleanup thread-specific resources. ***OnExit()*** is NOT called by ***Discard()***, since the thread has not been run in this case and there are no thread-specific data to destroy. ***virtual ~CThread(void)*** The destructor is protected to avoid thread object premature destruction. For this reason, no thread object can be static or stack-allocated. It is important to declare any ***CThread*** derived class destructor as *protected*.

Thread Life Cycle

Figure 2 shows a typical thread life cycle. The figure demonstrates that thread constructors are called from the parent thread. The child thread is spawned by the ***Run()*** function only. Then, the user-provided ***Main()*** method (containing code created by user) gets executed. The thread's destructor may be called in the context of either parent or child thread depending on the state of the thread at the moment when ***Join()*** or ***Detach()*** is called.

There are two possible ways to terminate a thread. By default, after user-provided ***Main()*** function return, the ***Exit()*** is called implicitly to terminate the thread. User functions can call ***CThread::Exit()*** directly. Since ***Exit()*** is a static method, the calling function does not need to be a thread class member or have a reference to the thread object. ***Exit()*** will terminate the thread in which context it is called.

The ***CThread*** destructor is *protected*. The same must be true for any user-defined thread class in order to prohibit creation of static or automatic thread objects. For the same reason, a thread object can not be destroyed by explicit *delete*. All threads destroy themselves on termination, detaching, or joining.

On thread termination, ***Exit()*** checks if the thread has been detached and, if this is true, destroys the thread object. If the thread has not been detached, the thread object will remain "zombie" unless detached or joined. Either ***Detach()*** or ***Join()*** will destroy the object if the thread has been terminated. One should keep in mind, that it is not safe to use the thread object after a call to ***Join()*** or ***Detach()*** since the object may happen to be destroyed. To avoid this situation, the ***CRef<CThread>*** can be used. The thread object will not be destroyed until there is at least one ***CRef*** to the object (although it may be terminated and scheduled for destruction).

In other words, a thread object will be destroyed when all of the following conditions are satisfied:

- the thread has been run and terminated by an implicit or explicit call to ***Exit()***
- the thread has been detached or joined
- no ***CRef*** references the thread object

Which thread will actually destroy a thread object depends on several conditions. If the thread has been detached before termination, the ***Exit()*** method will destroy it, provided there are no ***CRef*** references to the object. When joined, the thread will be destroyed in the context of a joining thread. If ***Detach()*** is called after thread termination, it will destroy the thread in the context of detaching thread. And, finally, if there are several ***CRef*** objects referencing the same thread, it will be destroyed after the last ***CRef*** release.

This means that cleaning up thread-specific data can not be done from the thread destructor. One should override ***OnExit()*** method instead. ***OnExit()*** is guaranteed to be called in the context of the thread before the thread termination. The destructor can be used to cleanup non-thread-local data only.

There is one more possibility to destroy a thread. If a thread has been created, but does not need to be run, one can use ***Discard()*** method to destroy the thread object without running it. Again, the object will not be destroyed until there are ***CRef***s referencing it.

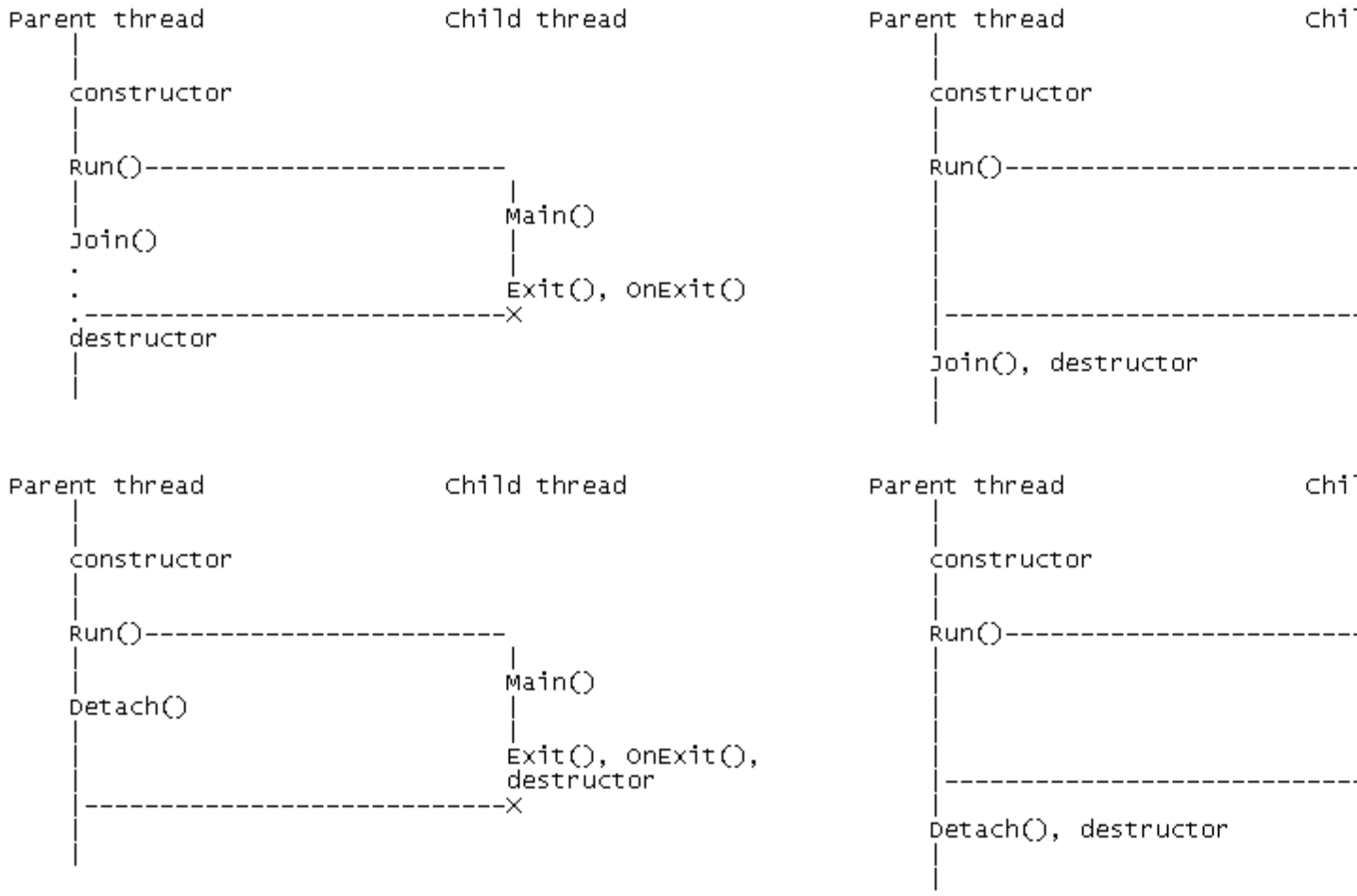

Figure 2: Thread Life Cycle

Referencing Thread Objects

It should be emphasized that regular (C) pointer to a thread object is not reliable. The thread may terminate at unpredictable moment, destroying itself. There is no possibility to safely access thread object after ***Join()*** using C pointers. The only solution to this problem is to use ***CRef*** class. ***CThread*** class provides a mechanism to prevent premature destruction if there are ***CRef*** references to the thread object.

Thread local storage (CTls<> class [%20])

The library provides a template class to store thread specific data: ***CTls<>***. This means that each thread can keep its own data in the same TLS object. To perform any kind of cleanup one can provide cleanup function and additional cleanup data when storing a value in the TLS object. The following example demonstrates the usage of TLS:

```
CRef< CTls<int> > tls;

void TlsCleanup(int* p_value, void* /* data */ )
{
    delete p_value;
```

```
}

...
void* CMyThread::Main()
{
    int* p_value = new int;
    *p_value = 1;
    tls->SetValue(p_value, TlsCleanup);
    ...
    p_value = new int;
    *p_value = 2;
    tls->SetValue(p_value, TlsCleanup);
    ...
    if (*tls->GetValue() == 2) {
        ...
    }
    ...
}
```

In the above example the second call to ***SetValue()*** will cause the ***TlsCleanup()*** to deallocate the first integer variable. To cleanup the last value stored in each TLS, the ***CThread::Exit()*** function will automatically call ***CTls<>::Reset()*** for each TLS used by the thread.

By default, all TLS objects are destroyed on program termination, since in most cases it is not guaranteed that a TLS object is not (or will not be) used by a thread. For the same reason the ***CTls<>*** destructor is protected, so that no TLS can be created in the stack memory. The best way of keeping TLS objects is to use ***CRef***.

Calling ***Discard()*** will schedule the TLS to be destroyed as soon as there are no ***CRef*** references to the object left. The method should be used with care.

Mutexes

The *ncbimtx.hpp* defines platform-independent mutex classes, ***CMutex***, ***CFastMutex***, ***CMutex-Guard***, and ***CFastMutexGuard***. These mutex classes are in turn built on the plaftform-dependent mutex classes ***SSystemMutex*** and ***SSystemFastMutex***.

In addition to the mutex classes, there are a number of classes that can be used for explict locks such as the ***CRWLock***, ***CAutoRW***, ***CReadLockGuard***, ***CWriteLockGuard*** and the platform-dependent read/write lock, ***CInternalRWLock***.

Finally, there is the ***CSemaphore*** class which is an application-wide semaphore.

These classes are discussed in the subsections that follow:

- CMutex
- CFastMutex
- SSystemMutex and SSystemFastMutex
- CMutexGuard and CFastMutexGuard
- Lock Classes

CMutex

The ***CMutex*** class provides the API for acquiring a mutex. This mutex allows nesting with runtime checks so recursive locks by the same thread is possible. This mutex checks the mutex owner before unlocking. ***CMutex*** is slower than ***CFastMutex*** and should be used when performance is less important than data protection. If performance is more important than data protection, use ***CFastMutex***, instead.

The main methods for ***CMutex*** operation are ***Lock()***, ***TryLock()*** and ***Unlock()***:

```
void Lock(void);
bool TryLock(void);
void Unlock(void);
```

The ***Lock()*** mutex method is used by a thread to acquire a lock. The lock can be acquired only if the mutex is unlocked; that is, not in use. If a thread has acquired a lock before, the lock counter is incremented. This is called nesting. The lock counter is only decremented when the same thread issues an ***Unlock()***. In other words, each call to ***Lock()*** must have a corresponding ***Unlock()*** by the same thread. If the mutex has been locked by another thread, then the thread must wait until it is unlocked. When the mutex is unlocked, the waiting thread can acquire the lock. This, then, is like a lock on an unlocked mutex.

The ***TryLock()*** mutex can be used to probe the mutex to see if a lock is possible, and if it is, acquire a lock on the mutex. If the mutex has already been locked, ***TryLock()*** returns *FALSE*. If the mutex is unlocked, than ***TryLock()*** acquires a lock on the mutex just as ***Lock()*** does, and returns *TRUE*.

The ***Unlock()*** method is used to decrease the lock counter if the mutex has been acquired by this thread. When the lock counter becomes zero, then the mutex is completely released (unlocked). If the mutex is not locked or locked by another thread, then the exception CMutexException (eOwner) is thrown.

The ***CMutex*** uses the functionality of ***CFastMutex***. Because ***CMutex*** allows nested locks and performs checks of mutex owner it is somewhat slower than ***CFastMutex***, but capable of protecting complicated code, and safer to use. To guarantee for a mutex release, ***CMutexGuard*** can be used. The mutex is locked by the ***CMutexGuard*** constructor and unlocked by its destructor. Macro `DEFINE_STATIC_MUTEX(id)` will define static mutex variable with name `id`. Macro `DECLARE_CLASS_STATIC_MUTEX(id)` will declare static class member of mutex type name `id`. Macro `DEFINE_CLASS_STATIC_MUTEX(class, id)` will define class static mutex variable `class::id`. The following example demonstrates usage of `CMutex`, including lock nesting:

```
static int Count = 0;
DEFINE_STATIC_MUTEX(CountMutex);

void Add2(void)
{
    CMutexGuard guard(CountMutex);
    Count += 2;
    if (Count < 20) {
        Add3();
    }
```

```
}

void Add3(void)
{
    CMutexGuard guard(CountMutex);
    Count += 3;
    if (Count < 20) {
        Add2();
    }
}
```

This example will result in several nested locks of the same mutex with the guaranteed release of each lock.

It is important not to unlock the mutex protected by a mutex guard. ***CFastMutexGuard*** and ***CMutexGuard*** both unlock the associated mutex on destruction. It the mutex is already unlocked this will cause a runtime error. Instead of unlocking the mutex directly one can use ***CFastMutexGuard::Release()*** or ***CMutexGuard::Release()*** method. This methods unlock the mutex and unlink it from the guard.

In addition to usual ***Lock()*** and ***Unlock()*** methods, the ***CMutex*** class implements a method to test the mutex state before locking it. ***TryLock()*** method attempts to acquire the mutex for the calling thread and returns *true* on success (this includes nested locks by the same thread) or *false* if the mutex has been acquired by another thread. After a successful ***TryLock()*** the mutex should be unlocked like after regular ***Lock()***.

CFastMutex

The ***CFastMutex*** class provides the API for acquiring a mutex. Unlike CMutex, this mutex does not permit nesting and does not check the mutex owner before unlocking. ***CFastMutex*** is, however, faster than ***CMutex*** and should be used when performance is more important than data protection. If performance is less important than data protection, use ***CMutex***, instead.

The main methods for ***CFastMutex*** operation are ***Lock()***, ***TryLock()*** and ***Unlock()***:

```
void Lock(void);
bool TryLock(void);
void Unlock(void);
```

The ***Lock()*** mutex method is used by a thread to acquire a lock without any nesting or ownership checks.

The ***TryLock()*** mutex can be used to probe the mutex to see if a lock is possible, and if it is, acquire a lock on the mutex. If the mutex has already been locked, ***TryLock()*** returns *FALSE*. If the mutex is unlocked, than ***TryLock()*** acquires a lock on the mutex just as ***Lock()*** does, and returns *TRUE*. The locking is done without any nesting or ownership checks.

The ***Unlock()*** method is used to unlock the mutex without any nesting or ownership checks.

The ***CFastMutex*** should be used only to protect small and simple parts of code. To guarantee for the mutex release the ***CFastMutexGuard*** class may be used. The mutex is locked by the ***CFastMutexGuard*** constructor and unlocked by its destructor. To avoid problems with initialization of static objects on different platforms, special macro definitions are intended to be used to

declare static mutexes. Macro `DEFINE_STATIC_FAST_MUTEX(id)` will define static mutex variable with name `id`. Macro `DECLARE_CLASS_STATIC_FAST_MUTEX(id)` will declare static class member of mutex type with name `id`. Macro `DEFINE_CLASS_STATIC_FAST_MUTEX(class, id)` will define static class mutex variable `class::id`. The example below demonstrates how to protect an integer variable with the fast mutex:

```
void ThreadSafe(void)
{
    static int Count = 0;
    DEFINE_STATIC_FAST_MUTEX(CountMutex);
    ...
    {{
        CFastMutexGuard guard(CountMutex);
        Count++;
    }}
    ...
}
```

SSystemMutex and SSystemFastMutex

The ***CMutex*** class is built on the plaftform-dependent mutex class, ***SSystemMutex***. The ***SSystemMutex*** is in turn built using the ***SSystemFastMutex*** class with additional provisions for keeping track of the thread ownership using the ***CThreadSystemID***, and a counter for the number of in the same thread locks (nested or recursive locks).

Each of the ***SSystemMutex*** and ***SSystemFastMutex*** classes have the ***Lock()***, ***TryLock()*** and ***Unlock()*** methods that are platform specific. These methods are used by the platform independent classes, ***CMutex*** and ***CFastMutex*** to provide locking and unlocking services.

CMutexGuard and CFastMutexGuard

The ***CMutexGuard*** and the ***CFastMutexGuard*** classes provide platform independent read and write lock guards to the mutexes. These classes are aliased as typedefs TReadLockGuard and TWriteLockGuard in the ***CMutexGuard*** and the ***CFastMutexGuard*** classes.

Lock Classes

This class implements sharing a resource between multiple reading and writing threads. The following rules are used for locking:

- if unlocked, the RWLock can be acquired for either R-lock or W-lock
- if R-locked, the RWLock can be R-locked by the same thread or other threads
- if W-locked, the RWLock can not be acquired by other threads (a call to ***ReadLock()*** or ***WriteLock()*** by another thread will suspend that thread until the RW-lock release).
- R-lock after W-lock by the same thread is allowed but treated as a nested W-lock
- W-lock after R-lock by the same thread results in a runtime error

Like ***CMutex***, CRWLock also provides methods for checking its current state: ***TryReadLock()*** and ***TryWriteLock()***. Both methods try to acquire the RW-lock, returning *true* on success (the RW-lock becomes R-locked or W-locked) or *false* if the RW-lock can not be acquired for the calling thread.

The following subsections describe these locks in more detail:

- CRWLock
- CAutoRW
- CReadLockGuard
- CWriteLockGuard
- CInternalRWLock
- CSemaphore

CRWLock

The ***CRWLock*** class allows read-after-write (R-after-W) locks for multiple readers or a single writer with recursive locks. The R-after-W lock is considered to be a recursive Write-lock. The write-after-read (W-after-R) is not permitted and can be caught when `_DEBUG` is defined. When `_DEBUG` is not defined, it does not always detect the W-after-R correctly, so a deadlock can occur in these circumstances. Therefore, it is important to test your application in the `_DEBUG` mode first.

The main methods in the class API are ***ReadLock()***, ***WriteLock()***, ***TryReadLock()***, ***TryWriteLock()*** and ***Unlock()***.

```
void ReadLock(void);
void WriteLock(void);
bool TryReadLock(void);
bool TryWriteLock(void);
void Unlock(void);
```

The ***ReadLock()*** is used to acquire a read lock. If a write lock has already been acquired by another thread, then this thread waits until it is released.

The ***WriteLock()*** is used to acquire a write lock. If a read or write lock has already been acquired by another thread, then this thread waits until it is released.

The ***TryReadLock()*** and ***TryWriteLock()*** methods are used to try and acquire a read or write lock, respectively, if at all possible. If a lock cannot be acquired, they immediately return with a *FALSE* value and do not wait to acqure a lock like the ***ReadLock()*** and ***WriteLock()*** methods. If a lock is successfully acquired, a *TRUE* value is returned.

As expected from the name, the ***Unlock()*** method releases the RW-lock.

CAutoRW

The ***CAutoRW*** class is used to provide a Read Write lock that is automatically released by the ***CAutoRW*** class' destructor. The locking mechanism is provide by a ***CRWLock*** object that is initialized when the ***CAutoRW*** class constructor is called.

An acquired lock can be released by an explict call to the class ***Release()*** method. The lock can also be released by the class destructor. When the destructor is called the lock if successfully acquired and not already released by ***Release()*** is released.

CReadLockGuard

The ***CReadLockGuard*** class is used to provide a basic read lock guard that can be used by other classes. This class is derived from the ***CAutoRW*** class.

The class constructor can be passed a ***CRWLock*** object on which a read lock is acquired, and which is registered to be released by the class destructor. The class's ***Guard()*** method can also be called with a ***CRWLock*** object and if this is not the same as the already registered ***CRWLock*** object, the old registered object is released, and the new ***CRWLock*** object is registered and a read lock acquired on it.

CWriteLockGuard

The ***CWriteLockGuard*** class is used to provide a basic write lock guard that can be used by other classes. The ***CWriteLockGuard*** class is similar to the ***CReadLockGuard*** class except that it provides a write lock instead of a read lock. This class is derived from the ***CAutoRW*** class.

The class constructor can be passed a ***CRWLock*** object on which a write lock is acquired, and which is registered to be released by the class destructor. The class's ***Guard()*** method can also be called with a ***CRWLock*** object and if this is not the same as the already registered ***CRWLock*** object, the old registered object is released, and the new ***CRWLock*** object is registered and a write lock acquired on it.

CInternalRWLock

The ***CInternalRWLock*** class holds platform dependent RW-lock data such as data on semaphores and mutexes. This class is not meant to be used directly by user applications. This class is used by other classes such as the ***CRWLock*** class.

CSemaphore

The ***CSemaphore*** class implements a general purpose counting semaphore. The constructor is passed an initial count for the semaphore and a maximum semaphore count.

When the ***Wait()*** method is executed for the semaphore, the counter is decremented by one. If the semaphore's count is zero then the thread waits until it is not zero. A variation on the ***Wait()*** method is the ***TryWait()*** method which is used to prevent long waits. The ***TryWait()*** can be passed a timeout value in seconds and nanoseconds:

```
bool TryWait(unsigned int timeout_sec = 0, unsigned int timeout_nsec = 0);
```

The ***TryWait()*** method can wait for the specified time for the semaphore's count to exceed zero. If that happens, the counter is decremented by one and ***TryWait()*** returns *TRUE*; otherwise, it returns *FALSE*.

The semaphore count is incremented by the ***Post()*** method and an exception is thrown if the maximum count is exceeded.

Working with File and Directories using CFile and CDir

An application may need to work with files and directories. The CORELIB provides a number of portable classes to model a system file and directory. The base class for the files and directories is ***CDirEntry***. Other classes such as ***CDir*** and ***CFile*** that deal with directories and files are derived form this base class.

The following sections discuss the file and directory classes in more detail:

- Executing a System Command using the System() Method
- Defining Spawned Process Modes (EMode type)
- Spawning a Process using SpawnX() Methods
- Waiting for a Process to Terminate using the Wait() method

CDirEntry class

This class models the directory entry structure for the file system and assumes that the path argument has the following form, where any or all components may be missing:

```
<dir><title><ext>

<dir>    =   file path             ("/usr/local/bin/"  or  "c:\windows\")
<title>   =   file name without ext ("autoexec")
<ext>    = file extension        (".bat" -- whatever goes after the last dot)
```

The supported filename formats are for the MS DOS/Windows, UNIX and MAC file systems.

The ***CDirEntry*** class provides the base methods such as the following for dealing with the components of a path name :

- ***GetPath()***: Get pathname.
- ***GetDir()***: Get the Directory component for this directory entry.
- ***GetBase()***: Get the base entry name without extension.
- ***GetName()***: Get the base entry name with extension.
- ***GetExt()***: Get the extension name.
- ***MakePath()***: Given the components of a path, combine them to create a path string.
- ***SplitPath()***: Given a path string, split them into its constituent components.
- ***GetPathSeparator()***: Get path separator symbol specific for the platform such as a '\' or '/'.
- ***IsPathSeparator()***: Check character "c" as path separator symbol specific for the platform.

- ***AddTrailingPathSeparator()***: Add a trailing path separator, if needed.
- ***ConvertToOSPath()***: Convert relative "path" on any OS to current OS dependent relative path.
- ***IsAbsolutePath()***: Note that the "path" must be for current OS.
- ***ConcatPath()***: Concatenate the two parts of the path for the current OS.
- ***ConcatPathEx()***: Concatenate the two parts of the path for any OS.
- ***MatchesMask()***: Match "name" against the filename "mask".
- ***Rename()***: Rename entry to specified "new_path".
- ***Remove()***: Remove the directory entry.

The last method on the list, the ***Remove()*** method accepts an enumeration type parameter, ***EDirRemoveMode***, which specifies the extent of the directory removal operation -- you can delete only an empty directory, only files in a directory but not any subdirectories, or remove the entire directory tree:

```
/// Directory remove mode.
enum EDirRemoveMode {
    eOnlyEmpty,     ///< Remove only empty directory
    eNonRecursive,  ///< Remove all files in directory, but not remove
                    ///< subdirectories and files in it
    eRecursive      ///< Remove all files and subdirectories
};
```

CDirEntry knows about different types of files or directory entries. Most of these file types are modeled after the Unix file system but can also handle the file system types for the Windows platform. The different file system types are represented by the enumeration type ***EType*** which is defined as follows :

```
/// Which directory entry type.
enum EType {
    eFile = 0,     ///< Regular file
    eDir,          ///< Directory
    ePipe,         ///< Pipe
    eLink,         ///< Symbolic link     (UNIX only)
    eSocket,       ///< Socket            (UNIX only)
    eDoor,         ///< Door              (UNIX only)
    eBlockSpecial, ///< Block special     (UNIX only)
    eCharSpecial,  ///< Character special
    //
    eUnknown       ///< Unknown type
};
```

CDirEntry knows about permission settings for a directory entry. Again, these are modeled after the Unix file system. The different permissions are represented by the enumeration type ***EMode*** which is defined as follows :

```
/// Directory entry's access permissions.
enum EMode {
    fExecute = 1,       ///< Execute permission
    fWrite   = 2,       ///< Write permission
    fRead    = 4,       ///< Read permission
    // initial defaults for dirs
    fDefaultDirUser  = fRead | fExecute | fWrite,
                        ///< Default user permission for dir.
    fDefaultDirGroup = fRead | fExecute,
                        ///< Default group permission for dir.
    fDefaultDirOther = fRead | fExecute,
                        ///< Default other permission for dir.
    // initial defaults for non-dir entries (files, etc.)
    fDefaultUser     = fRead | fWrite,
                        ///< Default user permission for file
    fDefaultGroup    = fRead,
                        ///< Default group permission for file
    fDefaultOther    = fRead,
                        ///< Default other permission for file
    fDefault = 8        ///< Special flag:  ignore all other flags,
                        ///< use current default mode
};
typedef unsigned int TMode;  ///< Binary OR of "EMode"
```

The directory entry permissions of read(r), write(w), execute(x), are defined for the "user", "group" and "others" The initial default permission for directories is "rwx" for "user", "rx" for "group" and "rx" for "others". These defaults allow a user to create directory entries while the "group" and "others" can only change to the directory and read a listing of the directory contents. The initial default permission for files is "rw" for "user", "r" for "group" and "r" for "others". These defaults allow a user to read and write to a file while the "group" and "others" can only read the file.

These directory permissions handle most situations but don't handle all permission types. For example, there is currently no provision for handling the Unix "sticky bit" or the "suid" or "sgid" bits. Moreover, operating systems such as Windows NT/2000/2003 and Solaris use Access Control Lists (ACL) settings for files. There is no provison in ***CDirEntry*** to handle ACLs

Other methods in ***CDirEntry*** deal specifically with checking the attributes of a directory entry such as the following methods:

- ***IsFile()***: Check if directory entry is a file.
- ***IsDir()***: Check if directory entry is a directory.
- ***GetType()***: Get type of directory entry. This returns an ***EType*** value.

- ***GetTime()***: Get time stamp of directory entry.
- ***GetMode()***: Get permission mode for the directory entry.
- ***SetMode()***: Set permission mode for the directory entry.
- ***static void SetDefaultModeGlobal()***: Set default mode globally for all CDirEntry objects. This is a class-wide static method and applies to all objects of this class.
- ***SetDefaultMode()***: Set mode for this one object only.

These methods are inherited by the derived classes ***CDir*** and ***CFile*** that are used to access directories and files, respectively.

CFile class

The ***CFile*** is derived from the base class, ***CDirEntry***. Besides inheriting the methods discussed in the previous section, the following new methods specific to files are defined in the ***CFile*** class:

- ***Exists()***: Check existence for a file.
- ***GetLength()***: Get size of file.
- ***GetTmpName()***: Get temporary file name.
- ***GetTmpNameEx()***: Get temporary file name in a specific directory and haveing a specified prefix value.
- ***CreateTmpFile()***: Create temporary file and return pointer to corresponding stream.
- ***CreateTmpFileEx()***: Create temporary file and return pointer to corresponding stream. You can additionally specify the directory in which to create the temporary file and the prefix to use for the temporary file name.

The methods ***CreateTmpFile()*** and ***CreateTmpFileEx()*** allow the creation of either a text or binary file. These two types of files are defined by the enumeration type, ***ETextBinary***, and the methods accept a parameter of this type to indicate the type of file to be created:

```
/// What type of temporary file to create.
enum ETextBinary {
    eText,          ///<Create text file
    eBinary         ///< Create binary file
};
```

Additionally, you can specify the type of operations (read, write) that should be permitted on the temporary files. These are defined by the enumeration type, ***EAllowRead***, and the ***CreateTmpFile()*** and ***CreateTmpFileEx()*** methods accept a parameter of this type to indicate the type operations that are permitted:

```
/// Which operations to allow on temporary file.
enum EAllowRead {
    eAllowRead,     ///< Allow read and write
    eWriteOnly      ///< Allow write only
};
```

CDir class

The ***CDir*** is derived from the base class, ***CDirEntry***. Besides inheriting the methods discussed in the ***CDirEntry*** section, the following new methods specific to directories are defined in the ***CDir*** class:

- ***Exists()***: Check existence for a directory.
- ***GetHome()***: Get the user's home directory.
- ***GetCwd()***: Get the current working directory.
- ***GetEntries()***: Get directory entries based on a specified mask parameter. Retuns a vector of pointers to ***CDirEntry*** objects defined by ***TEntries***
- ***Create()***: Create the directory using the directory name passed in the constructor.
- ***CreatePath()***: Create the directory path recursively possibly more than one at a time.
- ***Remove()***: Delete existing directory.

The last method on the list, the ***Remove()*** method accepts an enumeration type parameter, ***EDirRemoveMode***, defined in the ***CDirEntry*** class which specifies the extent of the directory removal operation -- you can delete only an empty directory, only files in a directory but not any subdirectories, or remove the entire directory tree.

CMemoryFile class

The ***CMemoryFile*** is derived from the base class, ***CDirEntry***. This class creates a virtual image of a disk file in memory that allow normal file operations to be permitted, but the file operations are actually performed on the image of the file in memory. This can result in considerable improvements in speed when there are many "disk intensive" file operations being performed on a file which is mapped to memory.

Besides inheriting the methods discussed in the ***CDirEntry*** section, the following new methods specific to memory mapped are defined in the ***CMemoryFile*** class:

- ***IsSupported()***: Check if memory-mapping is supported by the C++ Toolkit on this platform.
- ***GetPtr()***: Get pointer to beginning of data in the memory mapped file.

- ***GetSize()***: Get size of the mapped area.
- ***Flush()***: Flush by writing all modified copies of memory pages to the underlying file.
- ***Unmap()***: Unmap file if it has already been mapped.
- ***MemMapAdvise()***: Advise on memory map usage.
- ***MemMapAdviseAddr()***: Advise on memory map usage for specified region.

The methods ***MemMapAdvise()*** and ***MemMapAdviseAddr()*** allow one to advise on the expected uage pattern for the memory mapped file. The expected usage pattern is defined by the enumeration type, ***EMemMapAdvise***, and these methods accept a parameter of this type to indicate the usage pattern:

```
/// What type of data access pattern will be used for mapped region.
///
/// Advises the VM system that the a certain region of user mapped memory
/// will be accessed following a type of pattern. The VM system uses this
/// information to optimize work with mapped memory.
///
/// NOTE: Now works on UNIX platform only.
typedef enum {
    eMMA_Normal,      ///< No further special treatment
    eMMA_Random,      ///< Expect random page references
    eMMA_Sequential,  ///< Expect sequential page references
    eMMA_WillNeed,      ///< Will need these pages
    eMMA_DontNeed      ///< Don't need these pages
} EMemMapAdvise;
```

The memory usage advise is implemented on Unix platforms only, and is not supported on Windows platforms.

String APIs

The *ncbistr.hpp* file defines a number of useful constants, types and functions for handling string types. Most of the string functions are defined as class-wides static members of the class ***NStr***.

The following sections provide additional details on strinng APIs

- String Constants
- NStr Class
- UTF Strings
- PCase and PNocase

String Constants

For convenience, two types of empty strings are provided. A C-language style string that terminates with the null character ('\0') and a C++ style empty string.

The C-language style empty string constants are `NcbiEmptyCStr` which is a macro definition for the `NCBI_NS_NCBI::kEmptyCStr`. So the `NcbiEmptyStr` and `kEmptyCStr` are, for all practical purposes, equivalent.

The C++-language style empty string constants are `NcbiEmptyString` and the `kEmptyStr` which are macro definitions for the ***NCBI_NS_NCBI::CNcbiEmptyString::Get()*** method that returns an empty string. So the `NcbiEmptyString` and `kEmptyStr` are, for all practical purposes, equivalent.

The `SIZE_TYPE` is an alias for the string::size_type, and the `NPOS` defines a the constant that is returned when a substring search fails, or to indicate an unspecified string position.

NStr Class

The ***NStr*** class encapsulates a number of class-wide static methods. These include string concatenation, string conversion, string comparison, string search functions. Most of these string operations should be familiar to developers by name. Table 7 presents at a glance, a summary of these functions.

Table 7. NStr string functions

Function name	Parameters	Description
StringToNumeric	const string& str	Convert "str" to a (non-negative) integer value and return this value. Or return -1 if "str" contains any symbols other than [0-9], or if it represents a number that does not fit into an "int".
StringToInt	const string& str, int base = 10, ECheckEndPtr check = eCheck_Need	Convert specified string to int for the specified base. The check parameter determines whether trailing symbols (other than '\0') are permitted. The default is eCheck_Needed which means that if there are trailing symbols after the number, an exception will be thrown. If the value is eCheck_Skip, the string can have trailing symbols after the number.
StringToUInt	const string& str, int base = 10, ECheckEndPtr check = eCheck_Need	Similar to the ***StringToInt***, except that the conversion is to an unsigned int. Convert specified string to unsigned int for the specified base. The check parameter determines whether trailing symbols (other than '\0') are permitted. The default is eCheck_Needed which

Function name	Parameters	Description
		means that if there are trailing symbols after the number, an exception will be thrown. If the value is eCheck_Skip, the string can have trailing symbols after the number.
StringToLong	const string& str, int base = 10, ECheckEndPtr check = eCheck_Need	Similar to the ***StringToInt***, except that the conversion is to an long. Convert specified string to a long for the specified base. The check parameter determines whether trailing symbols (other than '\0') are permitted. The default is eCheck_Needed which means that if there are trailing symbols after the number, an exception will be thrown. If the value is eCheck_Skip, the string can have trailing symbols after the number.
StringToULong	const string& str, int base = 10, ECheckEndPtr check = eCheck_Need	Similar to the ***StringToLong***, except that the conversion is to an unsigned long. Convert specified string to an unsigned long for the specified base. The check parameter determines whether trailing symbols (other than '\0') are permitted. The default is eCheck_Needed which means that if there are trailing symbols after the number, an exception will be thrown. If the value is eCheck_Skip, the string can have trailing symbols after the number.
StringToDouble	const string& str, ECheckEndPtr check = eCheck_Need	Convert specified string to a double. The check parameter determines whether trailing symbols (other than '\0') are permitted. The default is eCheck_Needed which means that if there are trailing symbols after the number, an exception will be thrown. If the value is eCheck_Skip, the string can have trailing symbols after the number.
StringToInt8	const string& str	Convert specified string to a an Int8 value.
StringToUInt8	const string& str	Similar to the ***StringToInt8***, except that the conversion is to an unsigned eight byte integer. Convert specified string to a an UInt8 value.

Function name	Parameters	Description
StringToPtr	const string& str	Convert specified string to a void* pointer value.
IntToString	long value, bool sign = false	Convert specified long integer value to its string represenation. The sign parameter is used to determine whether the converted value should be preceded by the sign (+-) character.
UIntToString	unsigned long value	Convert specified unsigned long integer value to its string represenation.
Int8ToString	Int8 value, bool sign = false	Convert specified eight byte integer value to its string represenation. The sign parameter is used to determine whether the converted value should be preceded by the sign (+-) character.
UInt8ToString	Uint8 value	Convert specified eight byte unsigned integer value to its string represenation.
DoubleToString	double value	Convert specified double value to its string represenation.
DoubleToString	double value, unsigned int precision	Convert specified double value to its string represenation. The precision parameter specifies the precision value for conversion. If precision is more that maximum for current platform, then it will be truncated to this maximum.
DoubleToString	double value, unsigned int precision, char* buf, SIZE_TYPE buf_size	Convert specified double value to its string represenation and return the result of the conversion into the buf parameter. The precision parameter specifies the precision value for conversion. If precision is more that maximum for current platform, then it will be truncated to this maximum. The function returns the number of bytes stored in "buf", not counting the terminating '\0'.
PtrToString	const void* ptr	Convert specified void* pointer to its string represenation.
BoolToString	bool value	Convert the specified boolean to its string representation. Returns 'true' or 'false' string.
StringToBool	const string& str	Convert the specified string value to Boolean. Can recognize case-insensitive version as one of: 'true, 't', 'yes', 'y' for TRUE; and 'false', 'f', 'no', 'n' for FALSE. Returns TRUE or FALSE.

Function name	Parameters	Description
CompareCase	const string& str, SIZE_TYPE pos, SIZE_TYPE n, const char* pattern	Case-sensitive compare of a substring with a pattern. The substring to be compared is defined as starting from the 'pos' parameter and is 'n' characters long. The pattern to be matched is specified in the 'pattern' parameter.
CompareNocase	const string& str, SIZE_TYPE pos, SIZE_TYPE n, const char* pattern	Similar to ***CompareCase*** except that the comparison is case-insensitive. Case-insensitive compare of a substring with a pattern. The substring to be compared is defined as starting from the 'pos' parameter and is 'n' characters long. The pattern to be matched is specified in the 'pattern' parameter.
CompareNocase	const string& str, SIZE_TYPE pos, SIZE_TYPE n, const string& pattern	Similar to preceding ***CompareNocase*** except that the pattern is a string reference instead of a char*. Case-insensitive compare of a substring with a pattern. The substring to be compared is defined as starting from the 'pos' parameter and is 'n' characters long. The pattern to be matched is specified in the 'pattern' parameter.
CompareCase	const char* s1, const char* s2	Case-sensitive compare of two strings given the strings as char* values.
CompareNocase	const char* s1, const char* s2	Similar to ***CompareCase*** except that the comparison is case-insensitive. Case-insensitive compare of two strings given the strings as char* values.
CompareCase	const string& s1, const string& s2	Case-sensitive compare of two strings given the strings as string references.
CompareNocase	const string& s1, const string& s2	Similar to ***CompareCase*** except that the comparison is case-insensitive. Case-insensitive compare of two strings given the strings as string references.
Compare	const string& str, SIZE_TYPE pos, SIZE_TYPE n, const char* pattern, ECase use_case = eCase	Comparison of substring. The substring to be compared is defined as starting from the 'pos' parameter and is 'n' characters long. Whether to do a case sensitive compare(eCase -- default), or

Function name	Parameters	Description
		a case-insensitive compare (eNocase) is specified by the 'use_case' parameter.
Compare	const string& str, SIZE_TYPE pos, SIZE_TYPE n, const string& pattern, ECase use_case = eCase	Similar to preceding ***Compare*** except that the pattern is a string reference instead of a char*. Comparison of substring. The substring to be compared is defined as starting from the 'pos' parameter and is 'n' characters long. Whether to do a case sensitive compare(eCase -- default), or a case-insensitive compare (eNocase) is specified by the 'use_case' parameter.
Compare	const char* s1, const char* s2, ECase use_case = eCase	Compare of two strings given the strings as char* values. Whether to do a case sensitive compare(eCase -- default), or a case-insensitive compare (eNocase) is specified by the 'use_case' parameter.
Compare	const string& s1, const char* s2, ECase use_case = eCase	Similar to preceding ***Compare*** except that the first string is a string reference instead of a char*. Compare the two strings. Whether to do a case sensitive compare(eCase -- default), or a case-insensitive compare (eNocase) is specified by the 'use_case' parameter.
Compare	const char* s1, const string& s2, ECase use_case = eCase	Similar to preceding ***Compare*** except that the second string is a string reference instead of a char*. Compare the two strings. Whether to do a case sensitive compare(eCase -- default), or a case-insensitive compare (eNocase) is specified by the 'use_case' parameter.
Compare	const string& s1, const string& s2, ECase use_case = eCase	Similar to preceding ***Compare*** except that both strings are a string reference instead of a char*. Compare the two strings. Whether to do a case sensitive compare(eCase -- default), or a case-insensitive compare (eNocase) is specified by the 'use_case' parameter.
strcmp	const char* s1, const char* s2	Case sensitive compare of the two strings.
strcasecmp	const char* s1, const char* s2	Similar to preceding ***strcmp*** except that this is a case-insensitive compare. Case insensitive compare of the two strings.

Function name	Parameters	Description
strcmp	const char* s1, const char* s2, size_t n	Case sensitive compare of the two strings up to specified 'n' characters.
strncasecmp	const char* s1, const char* s2, size_t n	Similar to preceding ***strcmp*** except that this is a case-insensitive compare. Case insensitive compare of the two strings up to specified 'n' characters.
strftime	char* s, size_t maxsize, const char* format, const struct tm* timeptr	Formats specified time as string. This is a wrapper for the function strftime() that corrects handling %D and %T time formats on MS Windows.
ToLower	string& str	Convert string to lower case.
ToLower	char* str	Similar to preceding ***ToLower*** except that this uses a char* instead of a string reference. Convert string to lower case.
ToUpper	string& str	Convert string to uppercase.
ToUpper	char* str	Similar to preceding ***ToLower*** except that this uses a char* instead of a string reference. Convert string to uppercase.
StartsWith	const string& str, const string& start, ECase use_case = eCase	Check if a string starts with a specified prefix value. The 'start' parameter is the prefix value to check for. The 'use_case' parameter determines whether to do a case sensitive compare(default is eCase), or a case-insensitive compare (eNocase) while checking.
EndsWith	const string& str, const string& end, ECase use_case = eCase	Check if a string ends with a specified suffix value. The 'end' parameter is the suffix value to check for. The 'use_case' parameter determines whether to do a case sensitive compare(default is eCase), or a case-insensitive compare (eNocase) while checking.
Find	const string& str, const string& pattern, SIZE_TYPE start = 0, SIZE_TYPE end = NPOS, EOccurrence which = eFirst, ECase use_case = eCase	Finds the 'pattern' in the specified range of a string defined as starting from 'start' and ending with 'end'. The parameter 'which' when set to eFirst, means to find the first occurrence of the pattern and when set to eLast, this means to find the last occurrence. The parameter 'use_case' determines whether to do a case sensitive compare(default is

Function name	Parameters	Description
		eCase), or a case-insensitive compare (eNocase) while searching for the pattern.
FindCase	const string& str, const string& pattern, SIZE_TYPE start = 0, SIZE_TYPE end = NPOS, EOccurrence which = eFirst	Finds the 'pattern' in the specified range of a string defined as starting from 'start' and ending with 'end' doing a case sensitive search. The parameter 'which' when set to eFirst, means to find the first occurrence of the pattern and when set to eLast, this means to find the last occurrence.
FindNocase	const string& str, const string& pattern, SIZE_TYPE start = 0, SIZE_TYPE end = NPOS, EOccurrence which = eFirst	Finds the 'pattern' in the specified range of a string defined as starting from 'start' and ending with 'end' doing a case insensitive search. The parameter 'which' when set to eFirst, means to find the first occurrence of the pattern and when set to eLast, this means to find the last occurrence.
TruncateSpaces	const string& str, ETrunc where=eTrunc_Both	Truncate spaces in a string. The parameter 'which' controls which end of the string to truncate space from. Default is to truncate space from both ends (eTrunc_Both).
Replace	const string& src, const string& search, const string& replace, string& dst, SIZE_TYPE start_pos = 0, size_t max_replace = 0	Replace occurrences of a 'search' substring within the 'replace' string starting from 'start_pos' and return the result in 'dst'. The parameter 'max_replace' determines whether to replace no more than 'max_replace' occurrences of the substring. If 'max_replace' is zero(default), then replace all occurrences with 'replace'.
Replace	const string& src, const string& search, const string& replace, SIZE_TYPE start_pos = 0, size_t max_replace = 0	Replace occurrences of a 'search' substring within the 'replace' string starting from 'start_pos' and return the result in a new string. The parameter 'max_replace' determines whether to replace no more than 'max_replace' occurrences of the substring. If 'max_replace' is zero(default), then replace all occurrences with 'replace'.

Function name	Parameters	Description
Split	const string& str, const string& delim, list<string>& arr, EMergeDelims merge = eMergeDelims	Split a string using specified 'delim' delimiters and add the split tokens to 'arr' (a list of strings) and also return this array. The parameter 'merge' determines whether to merge the delimiters or not. The default setting of eMergeDelims means that delimiters that immediately follow each other are treated as one delimiter.
Tokenize	const string& str, const string& delim, list<string>& arr, EMergeDelims merge = eNoMergeDelims	Tokenize a string using specified 'delim' delimiters and add the tokens to 'arr' (a list of strings) and also return this array. The parameter 'merge' determines whether to merge the delimiters or not. The default setting of eNoMergeDelims means that delimiters that immediately follow each other are treated as separate delimiters.
SplitInTwo	const string& str, const string& delim, string& str1, string& str2	Split a string into two pieces 'str1' and 'str2' using the specified delimiters
Join	const list<string>& arr, const string& delim	Join strings in 'arr' using the specified delimiter.
PrintableString	const string& str, ENewLineMode nl_mode = eNewLine_Quote	Get a printable version of the specified string.
Wrap	const string& str, SIZE_TYPE width, list<string>& arr, TWrapFlags flags = 0, const string* prefix = 0, const string* prefix1 = 0	Wrap the specified string into lines of a specified 'width' and place these wrapped lines in 'arr' a list of strings. The 'flags' control how to wrap the words to a new line. The 'prefix' string is added to each wrapped line, except the first line, unless 'prefix1' is set. If 'prefix' is set to 0(default), do not add a prefix string to the wrapped lines. The 'prefix1' string is used for the first line. Use this for the first line instead of 'prefix'. If 'prefix1' is set to 0(default), do not add a prefix string to the first line.
Wrap	const string& str, SIZE_TYPE width, list<string>& arr, TWrapFlags flags = 0, const string* prefix , const string* prefix1 = 0	Similar to preceding ***Wrap*** except that only prefix1 is set to the default value (0). Wrap the specified string into lines of a specified 'width' and place these wrapped lines in 'arr' a list of strings. The 'flags' control how to wrap the words to a new line. The 'prefix' string is added to each wrapped line, except

Function name	Parameters	Description
		the first line, unless 'prefix1' is set. If 'prefix' is set to 0(default), do not add a prefix string to the wrapped lines. The 'prefix1' string is used for the first line. Use this for the first line instead of 'prefix'. If 'prefix1' is set to 0(default), do not add a prefix string to the first line.
Wrap	const string& str, SIZE_TYPE width, list<string>& arr, TWrapFlags flags = 0, const string* prefix , const string* prefix1 = 0	Similar to preceding ***Wrap*** except that neither prefix or prefix1 is set to the default value. Wrap the specified string into lines of a specified 'width' and place these wrapped lines in 'arr' a list of strings. The 'flags' control how to wrap the words to a new line. The 'prefix' string is added to each wrapped line, except the first line, unless 'prefix1' is set. If 'prefix' is set to 0(default), do not add a prefix string to the wrapped lines. The 'prefix1' string is used for the first line. Use this for the first line instead of 'prefix'. If 'prefix1' is set to 0 (default), do not add a prefix string to the first line.
WrapList	const list<string>& l, SIZE_TYPE width, const string& delim, list<string>& arr, TWrapFlags flags = 0, const string* prefix = 0, const string* prefix1 = 0	Wrap the specified list into lines of a specified 'width' and place these wrapped lines in 'arr' a list of strings. The 'flags' control how to wrap the words to a new line. The 'prefix' string is added to each wrapped line, except the first line, unless 'prefix1' is set. If 'prefix' is set to 0(default), do not add a prefix string to the wrapped lines. The 'prefix1' string is used for the first line. Use this for the first line instead of 'prefix'. If 'prefix1' is set to 0(default), do not add a prefix string to the first line.
WrapList	const list<string>& l, SIZE_TYPE width, const string& delim, list<string>& arr, TWrapFlags flags = 0, const string& prefix, const string* prefix1 = 0	Similar to preceding ***WrapList*** except that prefix1 is set to the default value and prefix is a reference to a string. Wrap the specified list into lines of a specified 'width' and place these wrapped lines in 'arr' a list of strings. The 'flags' control how to wrap the words to a new line. The 'prefix' string is added to each wrapped line, except the first line,

Function name	Parameters	Description
		unless 'prefix1' is set. If 'prefix' is set to 0(default), do not add a prefix string to the wrapped lines. The 'prefix1' string is used for the first line. Use this for the first line instead of 'prefix'. If 'prefix1' is set to 0(default), do not add a prefix string to the first line.
WrapList	const list<string>& l, SIZE_TYPE width, const string& delim, list<string>& arr, TWrapFlags flags = 0, const string& prefix, const string& prefix1	Similar to preceding ***WrapList*** except that neither prefix or prefix1 is set to the default value and both prefix and prefix1 are references to a string. Wrap the specified list into lines of a specified 'width' and place these wrapped lines in 'arr' a list of strings. The 'flags' control how to wrap the words to a new line. The 'prefix' string is added to each wrapped line, except the first line, unless 'prefix1' is set. If 'prefix' is set to 0(default), do not add a prefix string to the wrapped lines. The 'prefix1' string is used for the first line. Use this for the first line instead of 'prefix'. If 'prefix1' is set to 0(default), do not add a prefix string to the first line.

UTF Strings

The ***CStringUTF8*** class extends the C++ string class and provides support for Unicode Transformation Format-8 (UTF-8) strings.

This class supports constructors where the input argument is a string reference, char* pointer, and wide string, and wide character pointers. Wide string support exists if the macro `HAVE_WSTRING` is defined:

```
CStringUTF8(const string& src);
CStringUTF8(const char* src);
CStringUTF8(const wstring& src);
CStringUTF8(const wchar_t* src)
;
```

The ***CStringUTF8*** class defines assignment(=) and append-to string (+=) operators where the string assigned or appended can be a *CStringUTF8* reference, string reference, char* pointer, wstring reference, wchar_t* pointer.

Conversion to ASCII from ***CStringUTF8*** is defined by the ***AsAscii()*** method. This method can throw a StringException with error codes 'eFormat' or 'eConvert' if the string has a wrong UTF-8 format or cannot be converted to ASCII.

```
string AsAscii(void) const;
wstring AsUnicode(void) const;
```

PCase and PNocase

The ***PCase*** and ***PNocase*** structures define case-sensitive and case-insensitive comparison functions, respectively. These comparison functions are the ***Compare()***, ***Less()***, ***Equals()***, ***operator ()***. The ***Compare()*** returns an integer (-1 for less than, 0 for equal to, 1 for greater than). The ***Less()*** and ***Equals()*** return a TRUE if the first string is less than or equal to the second string. The ***operator()*** returns TRUE if the first string is less than the second.

A convenience template function AStrEquiv is defined that accepts the two classes to be compared as template parameters and a third template parameter that can be the comparison class such as the ***PCase*** and ***PNocase*** defined above.

Portable Time Class

The *ncbitime.hpp* defines ***CTime***, the standard Date/Time class which also can be used to represent elapsed time. Please note that the ***CTime*** class works for dates after 1/1/1900 and should not be used for elapsed time prior to this date. Also, since `Mac OS 9` does not support the daylight savings flag, ***CTime*** does not support daylight savings on this platform.

The subsections that follow discuss the following topics:

- CTime Class Constructors
- Other CTime Methods

CTime Class Constructors

The ***CTime*** class defines three basic constructors that accept commonly used time description arguments and some explict conversion and copy constructors. The basic constructors are the following:

- Constructor 1:

```
CTime(
  EInitMode          mode = eEmpty,
  ETimeZone          tz   = eLocal,
  ETimeZonePrecision
  tzp  = eTZPrecisionDefault);
```

- Constructor 2:

```
CTime(
  int year,
  int month,
  int day,
  int hour = 0,
  int minute = 0,
  int second = 0,
  long nanosecond = 0,
  ETimeZone tz  = Local,
  ETimeZonePrecision tzp = eTZPrecisionDefault);
```

- Constructor 3:

```
CTime(
  int year,
  int yearDayNumber,
  ETimeZone tz = eLocal,
  ETimeZonePrecision tzp = eTZPrecisionDefault);
```

In Constructor 1, the ***EInitMode*** is an enumeration type defined in the ***CTime*** class that can be used to specify whether to build the time object with empty time value(*eEmpty*) or current time (*eCurrent*). The ***ETimeZone*** is an enumeration type also defined in the ***CTime*** class that is used to specify the local time zone(*eLocal*) or GMT(*eGmt*. The ***ETimeZonePrecision*** is an enumeration type also defined in the ***CTime*** class that can be used to specify the time zone precision to be used for adjusting the daylight savings time. The default value(*eNone*) which means that daylight savings do not affect time calculatons.

Constructor 2 differs from Constructor 1 with respect to how the timestamp is specified. Here the time stamp is explictly specified as the year, month, day, hour, minute, second, nanosecond values. The other parameters of type ***ETimeZone*** and ***ETimeZonePrecision*** have the meanings discussed in the previous paragraph.

Constructor 3 allows the timestamp to be constructed as the Nth day (`yearDayNumber`) of a year(`year`). The other parameters of type ***EtimeZone*** and ***ETimeZonePrecision*** have the meanings discussed in the previous paragraph.

The explicit conversion constructor allows the conversion to be made from a string representation of time. The default value of the format string is `kEmptyStr` which implies that the format string has the format "M/D/Y h:m:s". As one would expect, the format specifiers, M, D, Y, h, m, s have the meanings month, day, year, hour, minute, second, respectively:

```
explicit CTime(
  const string& str,
  const string& fmt = kEmptyStr,
  ETimeZone tz = eLocal,
  ETimeZonePrecision tzp = eTZPrecisionDefault);
```

There is also a copy constructor defined that permits copy operations for ***CTime*** objects.

Other *CTime* Methods

Once the CTime object is constructed it can be accessed using the ***SetTimeT()*** and ***GetTimeT()*** methods. The ***SetTimeT()*** method is used to set the ***CTime*** with the timestamp passed by the ***time_t*** parameter. The ***GetTimeT()*** method returns the time stored in the ***CTime*** object as a ***time_t*** value. The ***time_t*** value measures seconds since January 1, 1900, so do not use these methods if the timestamp is less than 1900. Also time formats are in GMT time format.

A series of methods that set the time using the database formats ***TDBTimeI*** and ***TDBTimeU*** are also defined. These database time formats contain local time only and are defined as typedefs in *ncbitime.hpp*. The mutator methods are ***SetTimeDBI()*** and ***SetTimeDBU()***, and the accessor methods are ***GetTimeDBI()*** and ***GetTimeDBU()***.

You can set the time to the current time using the ***SetCurrent()*** method, or set it to "empty" using the ***Clear()*** method. If you want to measure time as days only and strip the hour, minute, second information you can use ***Truncate()*** method.

You can get or set the current time format using the ***GetFormat()*** and ***SetFormat()*** methods.

You can get and set the individual components of time such as year, day, month, hour, minute, second, nanosecond. The accessor methods for these components are named after the component itself and their meaning is obvious. For example, ***Year()*** for getting the year component, ***Month()*** for getting the month component, ***Day()*** for getting the day component, ***Hour()*** for getting the hour component, ***Minute()*** for getting the minute component, ***Second()*** for getting the second component, and ***NanoSecond()*** for getting the nanosecond component. The corresponding mutator methods for setting the individual components are the same as the accessor except that they have the prefix "Set" before them. For example, the mutator method for setting the day is ***SetDay()***. A word of caution on setting the individual components. You can easily set the timestamp to invalid values such as changing the number of days in the month of February to 29 when it is not a leap year, or 30 or 31.

A number of methods are available to get useful information from a ***CTime*** object. To get a day's year number (1 to 366) use ***YearDayNumber()***. To get the week number in a year, use ***YearWeekNumber()***. To get the week number in a month, use ***MonthWeekNumber()***. You can get the day of week (Sunday=0) by using ***DayOfWeek()***, or the number of days in the current month by using ***DaysInMonth()***.

There are times when you need to add months, days, hours, minutes, seconds to an existing ***CTime*** object. You can do this by using the ***AddXXX()*** methods where the "XXX" is the time component such as "Year", "Month", "Day", "Hour", "Minute", "Second", "NanoSecond" that is to be added to. Be aware that because the number of days in a month can vary adding months may change the day number in the timestamp. Operator methods for adding to (+=), subtracting from (-=), incrementing (++), decrementing (--) days are also available.

If you need to compare two timestamps, you can use the operator methods for equality (==), in-equality (!=), earlier than (<), later than (>), or a combination test such as earlier than or equal to (<=), or later than or equal to (>=).

You can measure the difference between two timestamps in days, hours, minutes, seconds, or nanoseconds. The timestamp difference methods have the form ***DiffXXX()*** where "XXX" is the time unit in which you want the difference calculated such as "Day", "Hour", "Minute", "Second", or "NanoSecond". Thus, ***DiffHour()*** can be used to calculate the difference in hours.

There are times when you may need to do a check on the timestamp. You can use ***IsLeap()*** to check if the time is in a leap year, or if it is empty by using ***IsEmpty()***, or if it is valid by using ***IsValid()***, or if it is local time by using ***IsLocalTime()***, or if it is GMT time by using ***IsGmtTime()***.

If you need to work with time zones explicitly, you can use ***GetTimeZoneFormat()*** to get the current time zone format, and ***SetTimeZoneFormat()*** to change it. You can use ***GetTimeZonePrecision()*** to get the current time zone precision and ***SetTimeZonePrecision()*** to change it. To get the time zone difference between local time and GMT, use ***TimeZoneDiff()***. To get current time as local time use ***GetLocalTime()***, and as GMT time use ***GetGmtTime()***. To convert current time to a specified time zone use ***ToTime()***, or to convert to local time use ***ToLocalTime()***.

Also defined for ***CTime*** are assignment operators to assign a ***CTime*** object to another ***CTime***, and an assignment operator where the right hand side is a time value string.

Template Utilities

The *ncbiutil.hpp* file defines a number of useful template functions, classes and struct definitions that are used in other parts of the library.

The following topics are discussed in this section:

- Function Objects
- Template Functions

Function Objects

The ***p_equal_to*** and ***pair_equal_to*** are template function classes that are derived from the standard ***binary_function*** base class. The ***p_equal_to*** checks for equality of objects pointed to by a pointer and ***pair_equal_to*** checks whether a pair's second element matches a given value. Another ***PPtrLess*** function class allows comparison of objects pointed to by a smart pointer.

The ***CNameGetter*** template defines the function ***GetKey()*** which returns the name attribute for the template parameter.

Template Functions

A number of inline template functions that make it easier to perform common operations on map objects are defined.

The ***NotNull()*** checks for a null pointer value and throws a ***CCoreException***, if a null value is detected. If the pointer value is not null, it is simply returned.

The ***GetMapElement()*** searches a map object for an element and returns the element, if found. If the element is not found it returns a default value which is usually set to 0 (null).

The ***SetMapElement()*** sets the map element. If the element to be set is null, it's existing key is erased.

The ***InsertMapElement()*** inserts a new map element.

The ***GetMapString()*** and ***SetMapString()*** are similar to the more general ***GetMapElement()*** and ***SetMapElement()*** except that they search a map object for a string. In the case of ***GetMapString()*** it returns a string, if found, and an empty string ("") if not found.

There are three overloads for the ***DeleteElements()*** template function. One overload accepts a container (list, vector, set, multiset) of pointers and deletes all elements in the container and clears the container afterwards. The other overloads work with map and multimap objects. In each case, they delete the pointers in the map object and clear the map container afterwards.

The ***AutoMap()*** template function works with a cache pointed to be an ***auto_ptr***. It retrieves the result from the cache and if the cache is empty it inserts a vaue into the cache from a specified source.

A ***FindBestChoice()*** template function is defined that returns the best choice (lowest score) value in the container. The container and scoring functions are specified as template parameters. The ***FindBestChoice()*** in turn uses the ***CBestChoiceTracker*** template class that uses the standard unary_function as it's base class. The ***CBestChoiceTracker*** contains the logic to record the scoring function and keep track of the current value and the best score.

Miscellaneous Types and Macros

The *ncbimisc.hpp* file defines a number of useful enumeration types and macros that are used in other parts of the library.

The following topics are discussed in this section:

- Miscellaneous Enumeration Types
- AutoPtr Class
- ITERATE macros
- Sequence Position Types

Miscellaneous Enumeration Types

The enum type ***EOwnership*** defines the constants *eNoOwnership* and *eTakeOwnership*. These are used to specify relationships between objects.

The enum type ***ENullable*** defines the constants *eNullable* and *eNotNullable*. These are used to specify if a data element can hold a null or not-null value.

AutoPtr Class

The *ncbimisc.hpp* file defines an ***auto_ptr*** class if the `HAVE_NO_AUTO_PTR` macro is undefined. This is useful in replacing the STL's ***std::auto_ptr*** for compilers with poor "auto_ptr" implementation. Section STL auto_ptrs discusses details on the use of ***auto_ptr***.

Another class related to the ***auto_ptr*** class is the ***AutoPtr*** class. The Standard ***auto_ptr*** class from STL does not allow the auto_ptr to be put in STL containers such as list, vector, map etc. Due to nature of how ownership works in an auto_ptr class, the AutoPtr's copy constructor and assignment operator modify the state of the source ***AutoPtr*** object as it transfers ownership to the target ***AutoPtr*** object.

A certain amount of flexibility has been provided in terms of how the pointer is to be deleted. This is done by passing a second argument to the ***AutoPtr*** template. This second argument allows the passing of a functor object that defines the deletion of the object. You can define "malloc" pointers in ***AutoPtr***, or you can use an ***ArrayDeleter*** template class to properly delete an array of objects using "delete[]". By default, the internal pointer will be deleted using the "delete" operator.

ITERATE macros

When working with STL container classes, it is common to use a for-statement to set up a loop to iterate through the elements in a container. For this reason the `ITERATE` and `NON_CONST_ITERATE` macros have been defined to sequence through the container elements. These macros are listed here as their code more clearly explains how they work:

```
#define ITERATE(Type, Var, Cont) \
    for ( Type::const_iterator Var = (Cont).begin(), NCBI_NAME2(Var,_end) = (Cont).end();  Var !=
NCBI_NAME2(Var,_end);  ++Var )

#define NON_CONST_ITERATE(Type, Var, Cont) \
    for ( Type::iterator Var = (Cont).begin();  Var != (Cont).end();  ++Var )
```

The difference between the ITERATE and NON_CONST_ITERATE is that the former uses a constant iterarator and the latter uses a non constant iterator.

The upper case versions of these macros is preferred by convention. Lower case versions of these macros are also defined but their use has been deprecated.

Sequence Position Types

The ***TSeqPos*** and and ***TSignedSeqPos*** are defined to specify sequence locations and length. ***TSeqPos*** is defined as an unsigned int and ***TSignedSqPos*** is a signed int that should be used only when negative values are a possibility for reporting differences between positions, or for error reporting -- though exceptions are generally better for error reporting.